Don’t Optimize Yet!

On an approach for making early performance evaluation usable to Software Engineering

Group E4-114	Supervisor
René Hansen	Bent Thomsen
Dennis Micheelsen	

The Faculty of Engineering and Science
University of Aalborg

Department of Computer Science

TITLE:
Don't Optimize Yet!

SUBTITLE:
On an approach for making early performance evaluation usable to Software Engineering

SEMESTER PERIOD:
DAT6,
1st February 05 - 15th June 05

PROJECT GROUP:
E4-114
René Hansen, rhansen@cs.aau.dk
Dennis Micheelsen, sak@cs.aau.dk

SUPERVISOR:
Bent Thomsen, bt@cs.aau.dk

NO. COPIES: 6

NO. PAGES: 91

Abstract:

There are three rules to performance optimizations: Don't optimize yet!, Don't optimize yet!, Don't optimize yet! Traditionally, the popular notion towards performance has been to ignore it until it could no longer be ignored. However, many companies have faced the consequence of this approach, finding that some performance problems are impossible to solve through code optimization alone. A discipline called Software Performance Engineering(SPE) tries to alleviate performance problems before they arise by modelling performance already at the design phase. However, SPE has never caught on in the mainstream Software Engineering practices - mainly because it relies on specialized performance staff. In this thesis we propose an approach making early performance considerations more amenable to Software Engineering. We have created a J2ME test application that derives performance characteristics of the target environment. The test application can be used to supply input estimates that would otherwise need to be supplied by a SPE specialist. The input estimates from the test application are used to annotate activity diagrams according to the UML^{TM} Profile for Schedulability, Performance, and Time which results in performance predictions of the system under development. We have investigated the validity of our approach by applying it to a test case, a J2ME application.

Preface

This thesis is written at the Department of Computer Science at the faculty of Engineering and Science. The thesis is written by group E4-114 in the field of Database and Programming Technologies. We wish to thank our supervisor Bent Thomsen for his input and many inspiring sessions throughout the work on this thesis.

Aalborg, June, 2005

Dennis Micheelsen	René Hansen

Contents

List of Tables

List of Figures

Listings

1 Introduction

Producing high quality software is, or at least should be, the overriding goal of any software development project.
A number of factors determine the software quality including [36, 44]:

- *maintainability*, the effort required to locate and fix errors in a program;
- *flexibility*, the effort required to modify an operational program;
- *reusability*, the extent to which a program (or parts of a program) can be used in other applications - related to the packaging and scope of the functions that the program performs;
- *Usability*; Effort required to learn and operate a program;
- *Efficiency*, the amount of computing resources required by a program to perform its function.

However, while the first four of the quality attributes can be targeted *proactively* through modular design, well-documented code and a host of best-practices for user interface design, the general literature on software engineering does not advocate the same tactic for the *efficiency*, i.e., performance, aspect. Rather, performance is handled *reactively*, that is, in response to performance problems arising. In essence, this means that performance problems are not discovered until the latter stages of the systems development process when actual executable code exist. Traditionally, the approach to alleviating performance problems has consisted of performing code optimization and tuning. This approach presents several problems, however. Code optimizations and tuning may make the code harder to understand and maintain, it may introduce new bugs that become even harder to resolve due to the added complexity [5]. Consider the following example, that computes the sum of two rows. A straightforward version might compute the row sum as seen in Listing 1.1 [39].

```java

double[] rowsum = new double[n];
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
    rowsum[i] += arr[i][j];
```

Listing 1.1: Row sum calculation

A more efficient version might calculate the row sum as seen in Listing 1.2 [39].

```java
double[] rowsum = new double[n];
for (int i=0; i<n; i++) {
  double[] arri = arr[i];
  double sum = 0.0;
  for (int j=0; j<m; j++)
    sum += arri[j];
  rowsum[i] = sum;
}
```

Listing 1.2: Optimized row sum calculation

The second example results in more efficient code, but also code that is harder to read and understand.

This small example illustrates the point that achieving efficiency by code optimization and tuning may afflict other quality attributes such as the maintainability and flexibility of the software.

As a consequence, the general golden rule has been to avoid any performance considerations until the latest possible point, exemplified by statements such as:

> "Ignore efficiency through most of the development cycle. Tune performance once the program is running correctly and the design reflects your best understanding of how the code should be structured. The needed changes will be limited in scope or will illuminate opportunities for better design." [4]

> "Changes in the system architecture to improve performance should as a rule be postponed until the system is being (partly) built. Experience shows that one frequently makes the wrong guesses, at least in large and complex systems, when it comes to the location of the bottlenecks critical to performance. To make correct assessments regarding necessary performance optimization, in most cases, we need something to measure." [20]

As it is, the cost of correcting error increases dramatically throughout the lifetime of a software project. An error in this instance denotes any quality problem that is discovered. A study at IBM [36], based on cost data from actual software projects, indicated that from the time of design, the cost of correcting an error increases 6.5 times just before testing commences; 15 times during testing; and finally, after release between 60-100 times. Moreover, [41] notes that many performance problems cannot be solved through code optimizations and tuning, but instead require extensive redesign to be alleviated. As a result, performance problems uncovered during late stages of the development process may result in schedule and resource overruns, damaged customer relations, lost income and even project failure.

It would be beneficial to discover performance problems at an earlier point where they can be corrected faster and more cost-effective. And indeed, approaches for doing so, exist. Software Performance Engineering (SPE) is an approach that targets performance problems at the analysis and design phases through performance modelling of critical use cases and the scenarios that describe them. However, SPE presents a number of problems. It requires special skill and knowledge of performance-related issues, including knowledge of an appropriate formalism (e.g., Execution Graphs and Queueing Network Models) in order to be able to construct and solve the performance models. Also, SPE is an inherently people dependant approach in that it requires a performance specialist to supply input, in the form of estimates, to the performance models. As a result, the accuracy of the predictions from the performance models depend exclusively on the quality of the estimates supplied by the performance specialist. This raises a question regarding the reliability of the resulting performance predictions.

A number of alternative approaches have emerged from SPE. These approaches can be largely divided into two: 1) approaches that provide alternative ways of modelling performance and 2) approaches applying SPE to specific application domains (e.g., performance analysis of client-server systems). However, like their SPE ancestor, these approaches require performance skills and are dependant on a performance specialist to provide input-estimates to the performance models.

Although SPE emerged in 1981 it has still not been incorporated into the practices of Software Engineering (SE) [27]. Proposals have been made to adapt the SE process models to accommodate SPE into the development effort and ease the integration of SPE into SE by educating developers in performance and by introducing more scientific, formal models into SE [27, 38]. However, we do not feel that such an approach is likely to be accepted into the main SE community. This would also require a reorganization of the computer science (CS) courses being taught at the majority of universities, as they do not include any required performance evaluation courses for CS undergraduates [21, 27]. Instead, we feel that early performance considerations to a much larger extent must be adapted into the existing SE paradigm.

In this, our master thesis, we propose an alternative to the SPE way of estimating performance. Motivated by the aforementioned problems with SPE we propose an approach that is easily integrated into the software engineering domain.

This is achieved by substituting input estimates supplied by a performance specialist to a performance model with a test application that derives performance characteristics of a target device. The test application can used to obtain input estimates which can be used as part of the design to establish if there are any performance problems in a system. Also, we wish to hide details about performance model creation and solution which would otherwise require special performance knowledge.

Our focus is on making performance evaluation of systems at the design stage *transparent* and *usable* to software engineers, as well as producing an approach that provides more *reliable* performance predictions and is cost-efficient to effecuate.

- Transparency is achieved by hiding details of performance modelling and solution to the developer. The creation and solution of performance models is done "behind the scenes" without intervention from a developer.
- The approach is usable, in that we base input estimates to performance models on actual measurements on the execution environment, thus dispensing with the need for a performance specialist to provide input estimates.
- The fact that the estimates of our approach rely on actual measurements should add to the reliability of the resulting predictions.
- Finally, the approach is cost-efficient in that it saves the costs of expensive SPE activities and specialized performance staff.

To support our approach, we will model key performance scenarios using UML activity diagrams according to the UML 2.0 Superstructure Specification [31] and use the UML$^{\text{TM}}$ Profile for Schedulability, Performance, and Time Specification (UML-SPT) [13]. The activity diagrams will be annotated with input estimates from our test application. Various UML tools (e.g., [11, 25, 29, 33]) support the export of annotated UML models into XML. This XML representation can then be used as input by performance analysis tools, solved, and be reported back into the UML tool. In this way, developers can receive performance estimates of a system without having to manually create and solve the needed performance models (i.e., not requiring knowledge of specific formalisms used for model construction and solution).

The focus of our investigation is on performance of J2ME applications. Mobile devices is a rapidly increasing market and more and more applications are being developed for mobile devices, supported by high-level programming languages such as C# and Java. Of course the device characteristics are very limited when compared to traditional computers. As a result companies may experience performance problems when moving to the J2ME platform. Particularly two areas of operation may cause performance problems in the J2ME domain, namely the use of persistent storage and network. This thesis is aimed at estimating the performance of applications that makes use of these “heavy-hitters”.

We investigate the validity of our approach by applying it to a end-to-end (J2EE-J2ME) application and measuring the accuracy of the resulting estimates.

The remainder of this thesis is organized into three parts. In the first part we describe the existing approaches to handling performance problems early in the development process. We give an account of their problems and present our solution to these problems. This part also contains a description of the performance component of the UML-SPT profile which will be used to attach performance annotations to activity diagrams describing the scenarios of our test case. In the second part we present our approach, apply it to our test case, and evaluate the validity of our approach based on the results obtained. In the final part of this thesis we conclude on our approach and discuss possible future work.

Part I

Related Work

2 Software Performance Engineering

Software Performance Engineering (SPE) is an approach specifically targeted towards constructing software systems that meet their performance requirements. SPE focuses on achieving good performance primarily by constructing performance models, beginning at the early stages of systems development. The performance models model the performance of the suggested architecture and high-level design and are refined throughout the development process as more detailed knowledge becomes available. The reason for beginning performance modelling at an early stage is to identify performance bottleneck at an early point where correction of performance problems is more cost-efficient.

The modelling process focuses on the subset of the system's use cases which are deemed critical from a performance perspective.

SPE is a systematic approach that proceeds according to a nine step plan described below and illustrated in figure 2.1.

1. *Assess performance risk*: An assessment of performance risks is done in order to determine the amount of SPE effort that is required for a particular project. Performance risks include familiarity with the type of system that is to be developed, new technologies, computer and network ressources required, and so forth. A high degree of performance risks implies a greater

deal of effort to be put into SPE activities.

2. *Identify critical use cases*. The critical use cases are those that are important to the operation of the system, or that are important to performance as seen by the user, i.e., the *responsiveness* of the system. In the UML, use cases are represented by use case diagrams

3. *Select key performance scenarios*. A use case typically consists of several scenarios. Scenarios are externally visible execution paths which can be expressed in the UML by either sequence-, activity-, or collaboration diagrams capturing the flow of messages being passed between objects. As it is, SPE uses an augmented form of sequence diagrams that includes extensions for decomposing a scenario into subscenarios, as well as constructs to express looping, alternation and optional execution. Typically only a subset of the scenarios constituting a use case will be important from a performance perspective. The objective is to identify these scenarios. Indications to which are the key scenarios are those that are executed frequently, or those that are critical to the perceived performance of the system.

4. *Establish performance objectives*. Quantitative performance objectives are defined for each selected scenario in order to evaluate the performance characteristics of the system.

   The steps 5 through 8 are repeated until there are no outstanding performance problems.

5. *Construct performance models*. SPE uses execution graphs to represent the software processing steps of each the selected performance scenarios. The sequence diagrams representing these key scenarios are translated to execution graphs.

6. *Determine software resource requirements*. Software resources are the resources affecting the performance of the software. The types of software resources to consider depend on the application to be developed and the software that the application interfaces with, so software resources may include CPU, I/O, messages, etc. as well as, say, calls to middleware functions or functions in different processes. Determining software resource requirements means establishing the amount of work that is required for each of these software resources, for instance how many messages must be sent or how many CPU instructions are needed to perform a given operation.

7. *Add computer resource requirements*. The computer resources represent key devices in the execution environment from which the identified software resources require service. Computer resources include CPU time, number of physical I/O, size of messages sent and so forth. Information about the execution environment can be obtained from the UML deployment diagram and

other documentation. The software resource requirements are mapped onto the amount of service they require from the computer resources. Computer resource requirements are represented in a so-called *overhead matrix*.

8. *Evaluate the models*. Solving the execution graph yields the resource requirements of the proposed software in isolation, i.e., without considering external factors, such as other programs competing for resources. If the solution indicates that there are no performance problems, the next step is to solve the *system execution model*. The system execution model characterizes the performance of the software in the presence of factors that could cause contention for resources, such as other programs competing for CPU time. The system execution model is represented by means of a Queueing Network Model (QNM).

9. *Verify and validate the models*: A continuous verification and validation of the constructed models occur in parallel with the actual construction and subsequent evaluation of the models. Verification and validation answers the following questions: “Are we building the model right” and “are we building the right model”, respectively. Model verification and validation are aimed at determining whether the model predictions provide an accurate reflection of the software’s performance as well as detect whether there are any model omissions.

The preceding nine steps describe SPE as it is approached in a single phase of the development cycle. These steps are applied iteratively for each subsequent phase of the development process, each cycle resulting in more refined models as the design evolves and more information becomes available.

Computer ressource requirements specifies the characteristics of the target execution environment such as the types of devices available, the quantity of each device, their speed, and so forth, and also connect the software resource requirements to their device usage. As mentioned, computer resource requirements are expressed via a so-called overhead matrix, of which we will now give an example. This example is taken from [41].

Device	CPU	Disk	Network
Quantity	1	1	1
Service Unit	KInstr.	Phys. I/O	Msgs.
WorkUnit	20	0	0
DB	500	2	0
Msgs	10	2	1
Service Time	0.00001	0.02	0.01

The values inserted into the table represent the processing overhead for a transaction in an ATM application. The concrete values are not important, as they are hypothetical [41], but what is important, however, is how these values are arrived at.

The names of the devices are in the first row, the second row shows the quantity of each device that is available, and the third row describes the unit of service

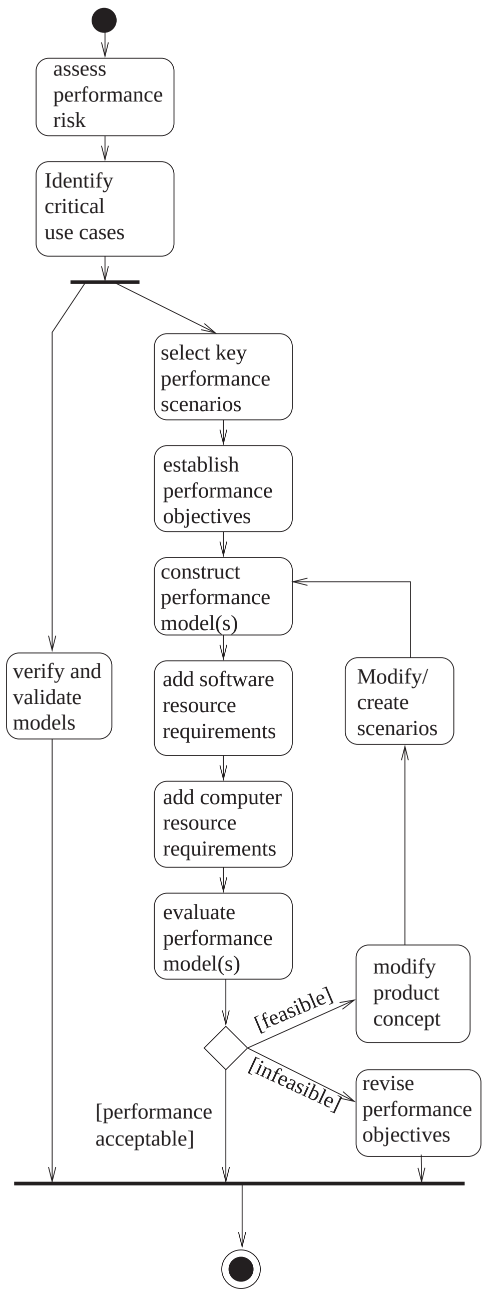

assess performance risk
Identify critical use cases
select key performance scenarios
establish performance objectives
construct performance model(s)
verify and validate models
add software resource requirements
Modify/ create scenarios
add computer resource requirements
evaluate performance model(s)
modify product concept
[feasible]
[infeasible]
revise performance objectives
[performance acceptable]

Figure 2.1: The SPE modelling process [41]

for each of the devices, e.g., a single unit for the CPU corresponds to 1000 CPU instructions.

The values in the center section of the table defines the mapping between software resource requests and device usage. For example, the `DB` row specifies that each DB request requires 500K CPU instructions and 2 physical I/Os. `Msgs` require 10K CPU instructions, 2 I/Os, and 1 network message. Finally, each `WorkUnit` requires 20K instructions.

The bottom row specifies the service time for each device.

An overhead matrix is created for each of the key performance scenarios, and this is used as input when creating the software execution model (Execution Graph) and system execution model (Queueing Network Model). Creating and solving both the Execution Graphs and later Queueing Network Models can be performed algorithmically and commercial tools exist for doing so. However, for the purpose of clarification, we present a brief description of this conversion process.

2.1 Execution Graphs

In order to calculate the accumulated time for a use case, which may consist of several performance scenarios, the sequence diagrams are converted to execution graphs and the accumulated time is then calculated. In the following we give a high-level description of the algorithms that perform this transformation. We start by giving a brief account of an execution graph.

2.1.1 Description

An execution graph consists of *nodes* connected by *arcs*. The nodes of an execution graph represent *processing steps*, which essentially are a collection of operations that perform a single function in the software system. Arcs represent the order of execution between processing steps.

Basic nodes represent processing steps at the lowest level of detail which mean that the processing step cannot be decomposed any further. The level of detail is relative to the current development stage, however, and a basic node may in a later stage be a candidate for further decomposition. Resource requirements are specified for each basic node. *Expanded nodes* on the other hand, represent processing steps that are elaborated in another subgraph. The subgraph may contain both basic and expanded nodes and can thus be recursively decomposed.

A *repetition node*, represented by a circle with a diagonal line, denotes a repetition of one or more nodes.

A *case node*, represented by a fork-like figure, denotes a conditional execution of alternate processing steps. It has one or more *attached nodes* that represent the choices that may be executed. The choices are annotated with their probability of being executed.

Figure 2.2 summarizes the various nodes.

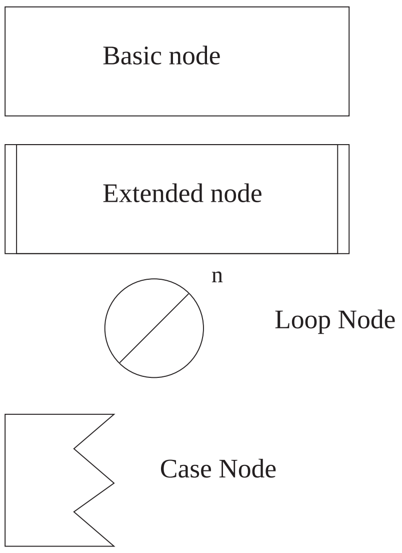


Figure 2.2: Basic execution graph nodes

2.1.2 Transforming sequence diagrams to execution graphs

Given that SPE uses an augmented form of sequence diagrams that allow extensions for decomposition, loops and conditional expressions the transformation process to execution graphs is straightforward. The most obvious approach to performing the transformation is to follow the message arrows through the sequence diagram, and make each action a basic node in the execution graph. However, in many cases, individual actions may be combined into a single basic node. Alternatively, an expanded node can be used to summarize a series of actions with the details shown in the subgraph.

2.1.3 Calculating the time for a use case

Once the performance scenarios constituting a use case have been converted to execution graphs, the accumulated time of the use case can be calculated. We now present a high-level description of the algorithms that perform this transformation.

The solution algorithms works as follows. The graph is examined and a basic structure is identified. There are three types of basic structures: sequence structures, denoting sequential execution, loop structures denoting repeated execution, and case structures, denoting conditional execution.

The solution algorithms utilize graph reduction to compute the time: The time for the structure is computed and replaced by a “computed node” that displays the time it takes to complete the structure. This process is continued until the graph is reduced to a single computed node, which represent the result of the analysis.

The computed time for a sequence structure is the sum of the times of the nodes in the sequence. This is illustrated in figure 2.3

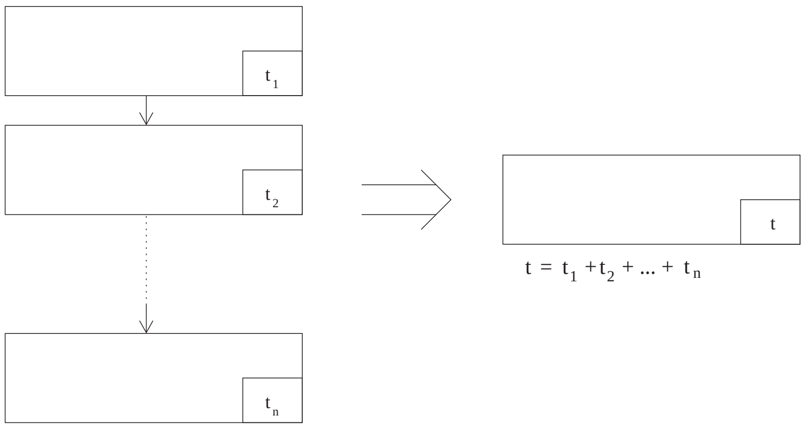


Figure 2.3: Sequence structure

For loop structures, the node time is multiplied by the loop repetition factor. This is illustrated in figure 2.4

The computation for case nodes is divided into three: shortest path, longest path and average analysis where the time for shortest path for the case node is the minimum of the times for the conditionally executed nodes, while the longest path represent the maximum (or worst case) times. For the average analysis, each node’s time is multiplied with its execution probability. Graph reduction for case nodes is illustrated in figure 2.5

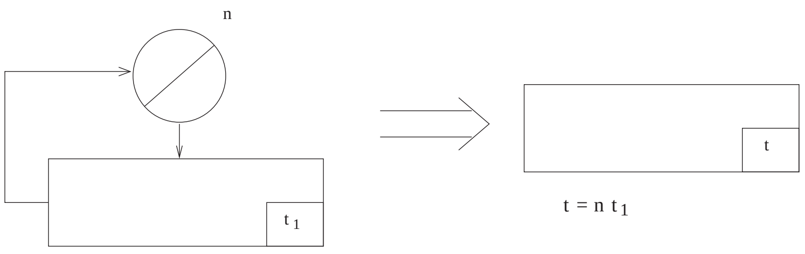


Figure 2.4: Loop structure

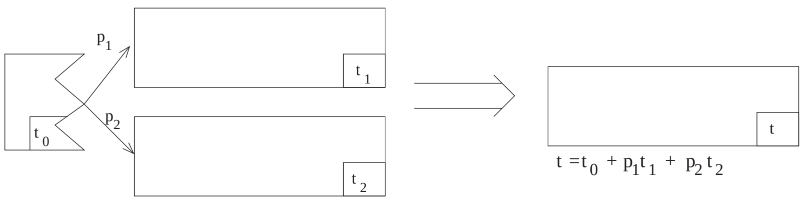


Figure 2.5: Case structure

For expanded nodes, the algorithms are applied recursively to the subgraph and the computed result substitutes the expanded node.

More advanced algorithms exist for parallel execution, but we will not discuss these, as they do not apply to the J2ME domain as of yet, and thus not relevant for the remainder of this thesis.

2.2 The System Execution Model

Solving the software execution model (execution graph) yields an analysis of the performance of the system in isolation. As such, the resulting estimate is an optimistic one. In order to characterize the performance of the system in the presence of other resources competing for resources, the next step is to create and solve a *system execution model* that takes into account a contention for resources. Possible sources of contention for resources include: multiple users of a system, other applications utilizing the same hardware resources, and an application may itself consist of several concurrent processes or threads.

SPE uses Queueing Network Models (QNM) to represent the system execution model. We will now give a brief introduction into QNMs. This description is based on [41, 45]

2.2.1 Queueing Network Models

Queueing Network Models (QNM) represents a system as a network of queues and servers. *Servers* represent resources (such as CPU, disk or network) that provide some service to the software and *queues* represent *jobs* (a generic term for computations requesting service) waiting for service from a server.

The most basic structure is a single server with an attached queue. A simple example illustrating the queue-server relationship is this: An application is started. To begin execution, the application (or job) requires service from a CPU (the server); if the server is busy, the job is held in a queue until the server is available. When the server is available, the job is removed from the queue and receives service from the server. Upon completion, the job leaves the server. Similarly, the queue-server relationship may represent a web service receiving a request, processing the request, and returning a response.

Of course, most systems utilize several different resources (CPU, one or more disks, network, etc.), so several queue-servers are connected in a network.

Figure 2.6 depicts a simple queue and server. The time spent in the queue is called the *wait time* and the time spent receiving service is called the *service time*. The accumulated time (wait time + service time) is called the *residence time*.

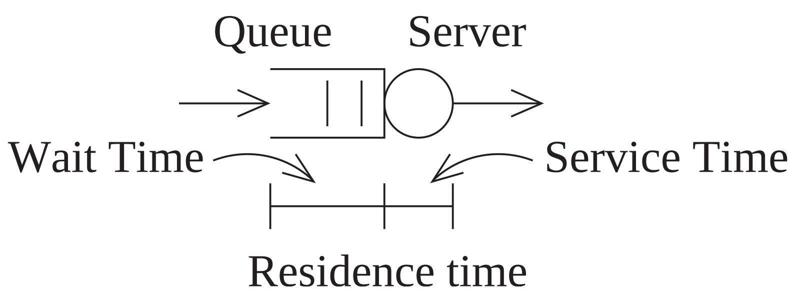


Figure 2.6: Queue-server representation of a resource [41]

Performance metrics

There are four average values that are calculated and used as the main performance metrics for each server. These are

- *Residence time*. The average residence time (wait time + service time) is calculated.
- *Utilization*, which is calculated as the average percent of the time that the server is busy supplying service.

- *Throughput*. This is the average rate of job completion.
- *Queue length*. This is the average number of jobs at a server. This includes both jobs in the queue as well as jobs receiving service.

The values of the above metrics are affected by factors such as the number of jobs, the amount of service they require, the time required for the server to service the jobs, and the scheduling policy that is used (First-Come-First-Served, time slicing, priority scheduling, etc.)

In order to determine these metrics, the server in question is observed over a period of time and the arrival and completion of jobs as well as the busy time for the server are measured. Then the four performance metrics are calculated as follows:

If we say that the measurement period is T, number of arrivals A, number of completions C, and busy time B then the utilization and throughput can be immediately calculated by using the measurement period T as divisor:

$$\text{Utilization } U = \frac{B}{T}$$

$$\text{Throughput } X = \frac{C}{T}$$

The *mean service time*, *S*, can also be calculated. The mean service time is the average amount of time that a job spends receiving service from the server:

$S = \frac{B}{C}$

In order to determine the residence time and queue length, the measurement period must be split up into intervals. Consider a hypothetical execution profile, taken from [41], and illustrated in fig 2.7.

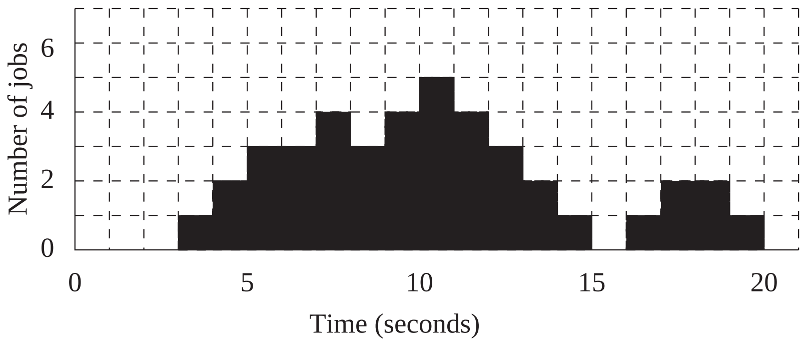


Figure 2.7: Hypothetical execution profile

This measurement period last 20 seconds and is divided into one second intervals. Then the number of jobs at the server for each interval, W, is sum up:

$$W = \sum_{intervals}(\#jobs)$$

In this example, this adds to:

$$W = 0+0+0+1+2+3+3+4+3+4+5+4+3+2+1+0+1+2+2+1 = 41$$

The residence time (R) and queue length N can now be calculated using W as numerator:

$$R = \frac{W}{C}$$

$$N = \frac{W}{T}$$

As seen, the above calculations of the main performance metrics rely on actual measurements. However, the approach of SPE in the early phases of the development process, is to rely on *estimates*. The estimates concern the predicted *workload intensity* and *service requirements*. Workload is a term denoting the collection of requests for service and workload intensity is a measure of the number of requests made in a given time interval. Service requirements are the amount of time that the workload requires from each of the devices.

Types of QNM

There are two types of QNMs: Closed models and open models.

Open models are appropriate for modelling systems where jobs arrive, receive some service, and then leave the system. An example might be a web service that receives a request, processes the request, and then sends a response. For an open system the *workload intensity* is specified as the *arrival rate*, that is, the rate at which jobs arrive for service. The service requirements are specified as the *number of visits* for each device and the *mean service time* per visit.

In contrast to open models, closed models have no external arrivals or departures. Instead, jobs keep circulating among queues. Closed models are appropriate for interactive systems where users issue a request, receive a response, and then issue another request.

Solving an open QNM The following parameters are specified:
λ is the arrival rate of jobs
V_i is the number of visits to a device i.
S_i is the mean service time at device i.

The performance metric can then be solved as follows:
The *system throughput* equals the arrival rate at the system assuming *job flow balance*. Job flow balance is a property stating that the system is fast enough to handle the arrivals.

System throughput, X_0

$$X_0 = \lambda$$

Device throughput, the throughput of a device i, denoted X_i is calculated by multiplying the system throughput with the number of visits to the device:

$$X_i = X_0 \times Vi$$

The *device utilization*, which is the utilization of a device i, U_i, is arrived at by multiplying the device throughput with the mean service time:

$$U_i = X_i \times S_i$$

The *device residence time* for each visit to a device i, R_i, is calculated using the mean service time and utilization of the device as follows:

$$R_i = \frac{S_i}{1-U_i}$$

The *device queue length* at a device i, N_i, is the device residence time multiplied with the device throughput:

$$N_i = R_i \times X_i$$

Finally, the system response time for the QNM as a whole is calculated. The calculation of the system response time uses Little's Formula [41] which also require the *average number of jobs* (N) in the entire QNM. This can be calculated as follows:

$$N = \sum_i N_i$$

Now the system response time, RT, can be calculated using Little's Formula as follows:

$$RT = \frac{N}{X_0}$$

Solving a closed QNM When solving a closed QNM the workload intensity is specified as the *number of users* (or simultaneous jobs) and the *think time*, which is the average delay between receiving a response and sending the next request. Solving a closed QNM is more complex and the description of this solution is beyond the scope of this thesis.

Creating and solving the system execution model

We will now give a brief description of the process of creating and solving a system execution model.

In order to construct and solve a system execution model, the first step is to add queue-servers to represent the key computer resources and devices. Then, connections between queues are added. The third step is to decide whether the system is modeled more appropriate using an open or closed QNM. Fourthly, the workload intensities for each scenario must be determined. The arrival rate and/or number of users is based on an anticipated use of the system and the service times are obtained from the solution to the software execution model. The last step of the process is to specify the service requirements. The device characteristics (processor speed, average time to complete an I/O operation, etc.) come from specifications of the execution environment while the visits and service times come from the software execution model.

2.3 Other approaches

A number of alternative approaches to early performance evaluation have emerged from the SPE foundation. These contributions can be largely divided into two categories: 1) Those that model performance by alternate means as compared to SPE, and 2) approaches targeted at specific application domains.

2.3.1 Alternate modelling approaches

[24] incorporates activity diagrams into the SPE approach. In addition, they present a prototype for a CASE tool that translates model elements from activity diagrams into a Generalized Stochastic Petri Net (GSPN).

[8] present a methodology that from UML use case diagrams, sequence diagrams and deployment diagrams produce an Extended Queueing Network Model (EQNM). The methodology includes algorithms for extracting performance aspects from the UML models and integrating them into the EQNM.

[9] takes a somewhat similar approach in that it relies on information contained in use case diagrams, sequence diagrams and deployment diagrams. The resulting performance model, however, is a QNM.

[35] propose a graph-grammar based method for transforming performance annotated UML models into a Layered Queueing Network (LQNM) performance model. The transformation takes as input an XML file that describes the UML model according to the XMI (XML Metadata Interchange) [30] interface and outputs a corresponding LQNM model description file. They rely on a combination of collaboration diagrams, deployment diagrams, and activity diagrams to express the software architecture, the allocation of software components to hardware resources, and represent the performance scenarios. They report on problems regarding the inter-operability with UML tools, since the UML tools do not entirely support the UML standard. As a consequence, they had to change the XML files exported from the UML tools in order to add missing features.

The authors of [35] extend their previous work in this article [14], proposing an alternative approach that implements the UML to LQN transformation by using XSLT (Extensible Stylesheet Language for Transformations) [46]. The input to the transformation is once again an XML file representation of the UML model and the output an LQNM description file. They provide a comparison with their previous approach concluding that the XSLT transformation approach is faster to develop. However, their problems with existing UML tools are unresolved.

Lastly, the authors have continued their work from [14,35] in [15]. It directly builds on [14] by using an XML representation of UML models according to the XMI interface, exported by a UML tool (Rational Rose). The transformation is done via XSLT, and this time the UML models are transformed into a simulation-based performance model (CSIM18 [43]). The work in all three articles is constrained to the client-server architectural pattern.

2.3.2 Specific application domain approaches

PRIMAmob-UML is an extension of PRIMA-UML [9] specifically targeted at systems that make use of mobile code. To this end PRIMAmob-UML uses an extended UML notation as well as extended EGs and EQNMs to account for mobile code.

[22] is an example of SPE being applied in the development process of a Digital Signal Processing a (DSP) application.

[6] applies SPE to web services. The paper has two contributions: 1) it proposes a web services based infrastructure to support Clinical Decision Support Systems (CDSSs) for processing data from multiple medical domains, and 2) it uses SPE to analyze the performance of the proposed system.

[26] present a methodology for evaluating the performance of the design of client-server systems. They base their methodology on a special software performance engineering language, developed by one of the authors, called `Clisspe` (CLIent/Server Software Performance Evaluation). A compiler for the Clisspe language generates a performance model (QNM) that can be solved by a model (QNM) solver.

[23] present a trace tool called EXTRA (Executable Trace Files) developed by IBM as part of the Visual Age for C++ product. They show how this tool can be used to generate traces for key performance scenarios of a prototype or a partially implemented systems in order to arrive at estimates which a performance analyst can then map into a Layered Queueing Network Model. These performance scenarios are then inserted into another IBM product named the Distributed Application Development Toolkit (DADT) which is used to consider the performance impact of design and configuration changes. These two tools are applied to a case study of a distributed application system.

3 Problems and proposed solutions

3.1 Problems with SPE

The greatest problem of adopting SPE as we view it, is the heavy reliance on a performance specialist being associated to a development project, in order to make SPE work.

This fact is probably expressed most clearly in step number 7 of the SPE approach: adding computer resource requirements. Adding computer resource requirements means determining the amount of service that is required for various operations.

The overhead matrix representing the computer resource requirements is what drives the model solutions, and thus result in the final estimates of the system.

But manually supplying the computer resource requirements constitutes a great problem, as we will now illustrate.

In order to determine the approximated time it takes to complete a key task (key performance scenario) the amount of CPU instructions must be estimated.

We argue that most system developers/programmers do not have a clear appreciation of how many CPU instructions correspond to an operation specified in a high-level programming language. For instance, few programmers probably realize that a database query may involve several hundred thousands or even millions of instructions [41]. In order to calculate a correspondence between application lines of code and CPU instructions we would need compilable code. Of course, this defers the whole point of estimating performance at an early point. For the sake of argument, let's assume that we do have compilable code. Even then, different compilers may generate significantly varying machine code or bytecode, and advanced compilers may perform optimizations that may be near impossible to predict beforehand.

There is also the problem of information hiding. When using middleware layers, interfaces or externally made components, such as when combining web services, the details of functionality are obscured. This is usually considered good practice from a software engineering perspective, but it significantly adds to the complexity of performance analysis.

As a consequence, any estimates which are to be likely predictions of the performance, requires the presence of a highly competent performance specialist who has a profound understanding of intricate details of the target environment and relevant software. Maintaining the qualities of such a performance specialist - keeping him up to date with the continuous and rapid emergence of new technologies is a job in itself. This also makes the SPE approach inherently people dependent. Maintaining this level of performance knowledge in several developers, to account for any people departures, is not a feasible option, at least not in smaller organizations.

3.1.1 Reliability of performance predictions

The reliance on a performance specialist also introduces an element of uncertainty concerning the reliability of the resulting performance estimates (predictions). The accuracy of the resulting performance estimates are a direct function of the level of expertise of the performance specialist who supplies the input estimates. If the input estimates supplied by the performance specialist are not reasonably accurate then nor will the resulting estimates from solving the performance models be. As [40] notes: “Model making requires possibly invalid assumptions”.

3.1.2 SPE and OO

Object-oriented (OO) systems present additional challenges for SPE. Performing an OO function is likely to involve numerous and complex interactions between many different objects making the interactions difficult to trace. This difficulty is further compounded by polymorphism [41]. Add to this the advent of exceptions which may cause sudden breaks in the control flow, and the unpredictable performance spikes incurred by the garbage collector kicking in. These factors contribute to making performance estimations of an OO system extremely difficult.

3.1.3 The cost of SPE

Initially, the input estimates may rely on guesses, and then as the development process progresses and more information becomes available SPE activities are conducted in order to reflect this new knowledge: input estimates are adjusted, performance models are modified or recreated and the models are verified and validated.

Developing a model manually can be a very labor intensive and error-prone process. Applications may use a host of different hardware and software resources, including objects, threads, middleware, operating system processes, network, databases and so forth. Performance modelling needs to consider all these aspects. Furthermore, it is a major challenge to ensure that the model remains in sync with an evolving design [23].

This brings up the issue of the cost of implementing SPE. [41] argues that the cost of SPE is minor relative to the overall project cost. According to [41] the cost of SPE at Lucent Technologies range from less than one percent of the total project budget for projects with a low performance risk to ten percent of the total budget for projects where the performance risk is very high. Arguably, the majority of companies do not have a project budget that is anywhere near that of Lucent Technologies and the percentage-wise cost for SPE may thus be far greater for smaller companies.

Moreover, SPE may be a “hard sale”. If an organization is not developing projects/software where performance constitutes a great risk, or has not encountered performance problems in previous projects they are probably not prone to make an investment in the adoption of SPE to account for the possible occurrence of problems in future projects.

3.1.4 Determining amount of SPE effort

Achieving high quality software is a complex mix of factors that vary between projects. Quality in software encompasses quality attributes such as maintainability, flexibility, reusability, efficiency, and determining which ones should be emphasized in a particular project depend on the application to be built and the customer who requested it [36]. High performance may not be the most important criteria for success, so including SPE activities into every project may not be relevant. [41] notes that the amount of SPE effort must be evaluated beforehand and if the performance risk is low, the SPE effort should reflect this. However, determining the correct amount of SPE effort requires experience and a knowledge of issues that may prove problematic from a performance perspective.

3.1.5 The need for quantitative requirements

According to the SPE approach, quantitative requirements should be established for all key performance scenarios. This information should be supplied by a system architect in collaboration with a performance engineer as well as a marketing representative and/or a user representative.

As we see it, the demand for establishing quantifiable requirements present a problem as well. The customer (represented be a user representative) may very well be a non-technical person who does not have an appreciation of the amount of work behind each processing step, and will maybe accept no less than the ideal. The customer may not have a clear appreciation of how long say, 2 seconds waiting time is. This may sound acceptable to the customer (representant), but may prove as an unacceptable waiting time to actual end users. Moreover, if we take the performance engineer out of the equation for one minute, "ordinary" system developers may not have a clear understanding at this point either of the amount of work (in terms of exact, or even approximate, time measures) that is required for various parts of the system. Thus, they may inadvertently commit to response times, that may later prove impossible to obtain.

This situation clearly illustrates the necessity of a performance engineer being present who hopefully have enough experience to provide reasonably realistic estimates, and who is able to convey the arguments of why various parts take as long time as they do, in order to convince the customer.

If it turns out that the specified objectives can not be met, SPE states that an alternative may be to renegotiate the objectives with the customer. However, we find that such a tactic may damage customer relations, particularly if this situation occurs repeatedly throughout the development process. Even if the customer complies to the new objectives, we feel such a tactic may leave an impression of unprofessional conduct.

Lets say that the system to be built should start by collecting data from a server. This may be an action that should be performed once a day, say at the start of the work day. It may be the case that this fetching time really isn't that important, if the end user can use the waiting time to get or cup of coffee or something. In such a case, committing to a fetch time of say 10 seconds may be overly constrained, when in fact the user could easily tolerate, say, one minute, where the user is occupied doing other things. The process of trying to achieve these 10 seconds may prove impossible, necessitating a renegotiation, when in fact this could be avoided by listening to what is the typical workday scenario of the end user is.

3.1.6 Software Engineering and SPE

SPE has not yet been incorporated into the practices of Software Engineering (SE) [27]. The author of [27] point to a lack of scientific principles and models in SE, a lack of education in performance, and an IT workforce employing many people with less than a bachelors degree and no formal training as possible causes as to why performance is not considered at the design time in software engineering projects.

[21] has conducted a survey of undergraduate computer science courses at highly ranked computer science schools in the United States (the top 24 research and doctorate schools according to data from Computer Research Association's Taulbee Survey [21]) in order to find out what kind of education undergraduates are receiving with respect to designing, building, and maintaining software that exhibits good performance. The survey was aimed at the undergraduate level, since they represent the majority of the workforce - only about 20 percent of the computer scientist students receive a graduate degree.

The survey indicated a number of shortcomings, including:

- Performance was rarely discussed (64.87% of the software engineering courses spend little or no time discussing performance-related subjects)
- In the cases where performance *was* discussed, the subject was either inadequately or not defined (84.84% of the courses had an inadequate or missing performance definition)
- Performance was regarded as a late life cycle activity (72% of the courses advocated a reactive strategy to performance).
- In most cases performance was viewed as a low-level system or algorithmic problem (74.07% of the courses regarded low-level system components or algorithmic efficiency as having the greatest impact on application performance.)
- Not a single course discussed how to measure resource utilization (no techniques for measuring throughput and utilization of resources)
- Only few of the courses (27.02%) discussed performance modelling. Of these courses, half used models for performance validation before implementation while the other half used models to diagnose problems in an implemented system.

- SPE was only mentioned in two courses, and one of these incorrectly implied that SPE was a specialized discipline for database-centered and real-time applications.

Since performance issues have traditionally been considered a late life cycle activity, SPE is not reflected in the mainstream process models [38]. [38] point to a number of problems in the mainstream process models that adversely effects the integration of SPE into SE and propose an extension that explicitly takes SPE activities into account.

However, integrating SPE into SE is contingent on the appropriate amount of performance knowledge being available, which the survey [21] shows may far from always be the case.

As a result, achieving widespread use of SPE would entail a reorganization of the computer science courses being taught at the majority of universities, as well as a new or extended process model that explicitly takes SPE into account.

We do not regard these two conditions to be realistically met in the near future and as a consequence early performance considerations must rather be be adapted into the existing SE paradigm instead of vice versa.

3.2 Proposed solutions

We propose an alternative to the SPE way of estimating performance. Motivated by the problems with SPE, described in the previous section, we propose an approach, that is easily integrated into the software engineering domain.

This approach is realized by making a test application that captures performance characteristics of the execution environment. The user (developer) uses the test application to obtain performance measurements of a series of relevant operations executed on the target device. The results from these tests are then used as input to a performance annotated UML activity diagram. In order to remain compliant with the recent advances in UML performance annotations, we rely on the UML^{TM} Profile for Schedulability, Performance, and Time Specification (UML-SPT) for annotating the activity diagrams. The annotated diagrams can be exported into XML, and the XML representation of the model can then be used as input to a performance analysis tools which generates and solves an appropriate performance model and reports the result back into the appropriate UML tool.

Our objective is to make performance evaluation of systems in the design phase:

Usable in that developers do not require extensive knowledge of the multitude of different devices that applications can be developed to, nor knowledge of the amount of processing that various high-level operations incur on the underlying architecture.

Transparent by hiding details of performance model construction and solution from the developer.

Reliable by removing the issue of whether the input estimates supplied by the performance specialist can be trusted. By using a test application instead we should be able to obtain consistently reliable estimates.

Cost-efficient as costs associated with manually constructing, solving and later updating or recreating performance models, as well as costs associated with specialized performance staff, are saved.

Collectively, these advantages clearly make early performance validation significantly more amenable to incorporation into the mainstream practices of software engineering.

Regarding the issue of establishing quantitative requirements for all key performance scenarios that SPE advocates, we advice against it. Performance has traditionally been treated as a non-functional requirement [27, 36, 44], meaning that it is not directly concerned with the specific functions delivered by the system (i.e., not expressed as an explicit requirement). Rather, non-functional requirements relate to the system as a whole and are implicitly assumed to be satisfactory (e.g., showing satisfactory performance). By making performance a functional requirement we further rely on a skillful performance specialist (to establish reasonable requirements) which is the situation we are trying to avoid.

4

UML™ Profile for Schedulability, Performance and Time

The UML^{TM} profile for Schedulability, Performance, and Time [13] (UML-SPT) is a UML profile that extends the standard UML with notions for modelling schedulability, performance, and time. The point of interest for this thesis is the performance component of the profile which is used for general performance analysis of UML models. Figure 4.1 illustrates the entire profile graphically, and the area enclosed by a solid line indicates the place of the performance component in the UML-SPT profile. The main source of information on the performance component is derived from [13].

The performance component of the profile provides facilities for:

- Capturing quantitative performance *requirements*.
- Associating performance-related *Quality of Service(QoS) characteristics* with selected elements of an UML model.
- *Execution parameters* to allow modelling tools to compute predicted performance characteristics.

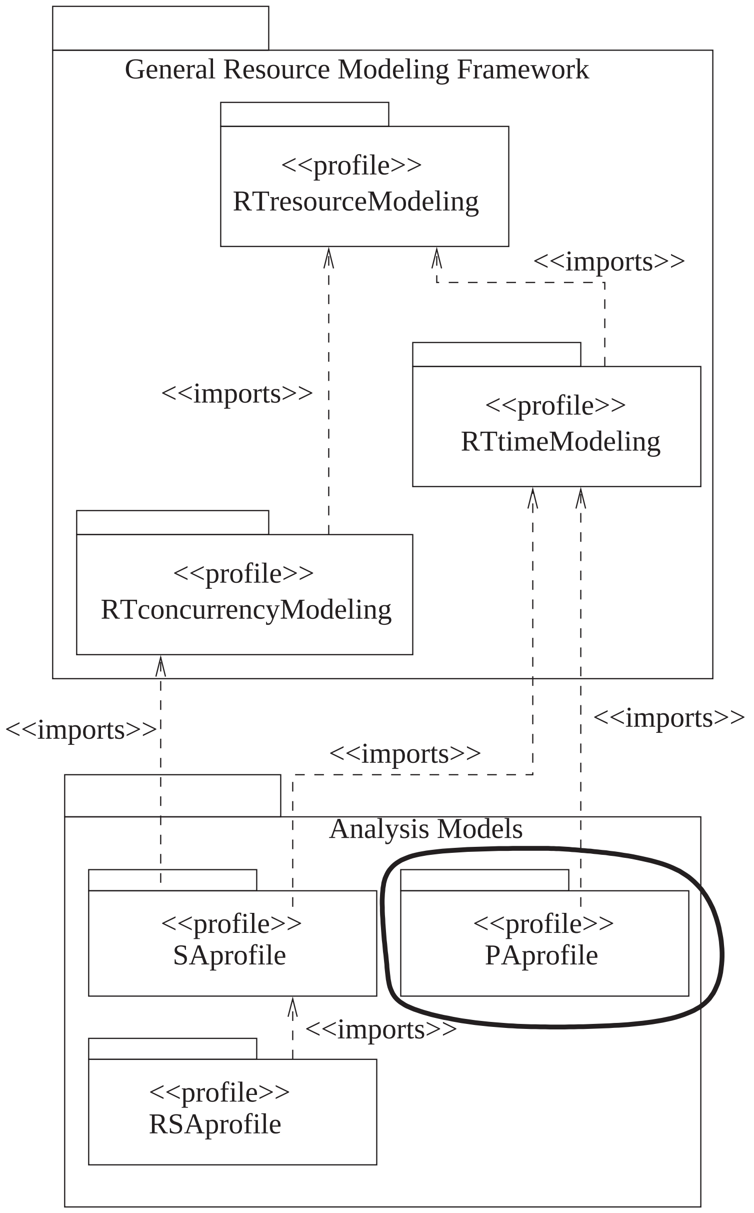


Figure 4.1: The UML profile for Schedulability, Performance, and Time. The performance component is indicated by the solid line.

- Presenting performance results computed by the tools.

The performance profile extends the UML metamodel with *stereotypes*, *tagged values* and *constraints*, which enables performance annotations to be attached to a UML model. [35]

A **Stereotype** is a construct that allows the creation of new model elements that are specific to a particular domain, such as performance. A stereotype is represented as a string enclosed in guillemets («»). An example might be a stereotype «processor »that indicates that this model element represents a processor.

A **tagged value** consists of a tag and value, that allows the specification of additional properties for a model element. The tag is the name of the property and the value field is the value for that property. The UML-SPT profile defines a formal language, called Tagged Value Language (TVL), for specifying the value. TVL is based on a very limited subset of the Perl programming language and the syntax should be familiar to most programmers (we refer to Appendix A of [13] for a description of TVL). A tagged value is enclosed by braces and appears as follows: {<tag-name> = <TVL-expression>} . An example might be a tagged value {processorSpeed = 1800Mhz} that is attached to a model element stereotyped by the aforementioned «processor »stereotype, indicating that the processor has a processor speed of 1800 Mhz.

A **constraint** is a restriction or condition which may be placed on an individual model element or a collection of elements. The constraint is enclosed by braces ({}) and interpreted according to a language which may be: the UML Object Constraint Language; a programming language (e.g., C++ or Java); a formal notation; or a natural language [41]. An example of a constraint may be this: {responseTime < 200}.

In the performance domain, stereotypes and tagged values can be used to capture information about the execution environment (e.g., processor speed, network speed, etc) while constraints can be used to specify performance requirements.

4.1 Performance analysis of a UML model

In order to conduct performance analysis of an annotated UML model, the UML model must be translated into a performance model. Then, a performance analysis tool can be used to solve the performance model. (Examples of performance tools can be found here [7, 18, 34, 37, 42, 43]). Finally the performance analysis results must be imported back into the UML model. This procedure is illustrated in figure 4.2

Performance tools may use different formalisms for performance analysis, including QNM, Layered QNM, Stochastic Process Algebra, Petri Nets, Markov Chains, and simulation models. However, as it is, we have not been able to find any performance analysis tools that, at this point, support the transformation process of anno-

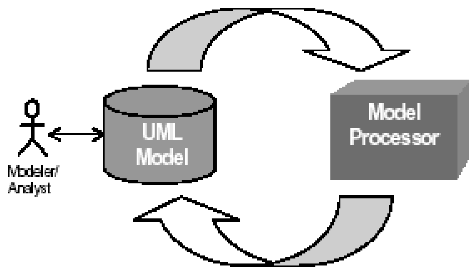


Figure 4.2: Abstract view of the model processing process [13]

tated UML models into input to the performance analysis tools. Rather, the tools we have cited above rely on a user supplying input manually. Granted, attempts have been made to try and automate, at least the first step, of the transformation process [8, 14, 15, 24, 35, 49] but the work has not resulted in tools for general use. The backward process (supplying the results of performance solutions back into the UML model) appear to be uninvestigated [35] at this point, meaning that the performance modelling of UML diagrams still require knowledge of creating and solving the appropriate performance models.

4.2 Concepts and techniques

The concepts and techniques of performance analysis in the UML-SPT realm resemble those which we have already introduced in Chapter 2. A *scenario* defines an execution path which has a response time and throughput. Each scenario is executed by a *workload* (collection of requests) with an applied *workload intensity*. An *open* workload has a stream of requests (or jobs) that arrive in a predefined pattern in which case the workload intensity denotes the arrival rate of the requests, whereas a *closed* workload has a fixed number of jobs that cycle between executing the scenario and spending an external delay period (think time) between receiving a response and issuing the next request. In this case the workload intensity is the number of users/simultaneous jobs and the *think time*.

A scenario consists of a number of *scenario steps* that are joined in a sequence with a predecessor-successor relationship. Steps may be connected via forks, joins and loops and a step may thus have multiple predecessors and successors. A step may be an elementary operation that cannot be decomposed any further or may be an operation that is defined by a sub-scenario.

To each step is assigned a mean execution count which is a measure of the average amount of times the step is repeated when it is executed. A step also has a *host execution demand* which is the execution time of the step on its host device in a given deployment.

The *resource demands* by a step include its host execution demand as well as the demands of all its sub-steps.

Resources are modelled as *servers*. *Resource-operations* of a resource are the steps, or sequence of steps which use the resource, consisting of the stages of acquiring, utilizing and releasing the resource. Resource-operations are thus similar to the *queue-server* relationship introduced earlier.

The *service time* of a resource is defined as the *host execution demand* of the steps hosted by the resource.

Performance measures for a system include resource utilizations, waiting times, execution demands (what SPE calls service requirements, e.g., CPU cycles), and the response times to execute a scenario or scenario step. Each measure may be defined in different versions:

Required value : A performance objective established for a scenario of scenario step. This value may originate directly from system requirements or from a performance estimate based on the requirements.

Assumed value : This value is supplied based on experience.

Estimated value : This value is calculated by a performance tool and reported back into the model.

Measured value : This value is obtained by conducting actual measurements.

The value of any of the above versions may be stated as one of several statistical properties, for instance, the mean or maximum value.

4.2.1 Domain model

Figure 4.3 shows the general performance model that identifies the basic abstractions and relationships used in performance analysis in the UML realm.

Performance context

A common usage of performance analysis tools is to analyze the system under varying conditions with different parameters while maintaining the same system structure. This way, the performance of the system can be estimated, accounting

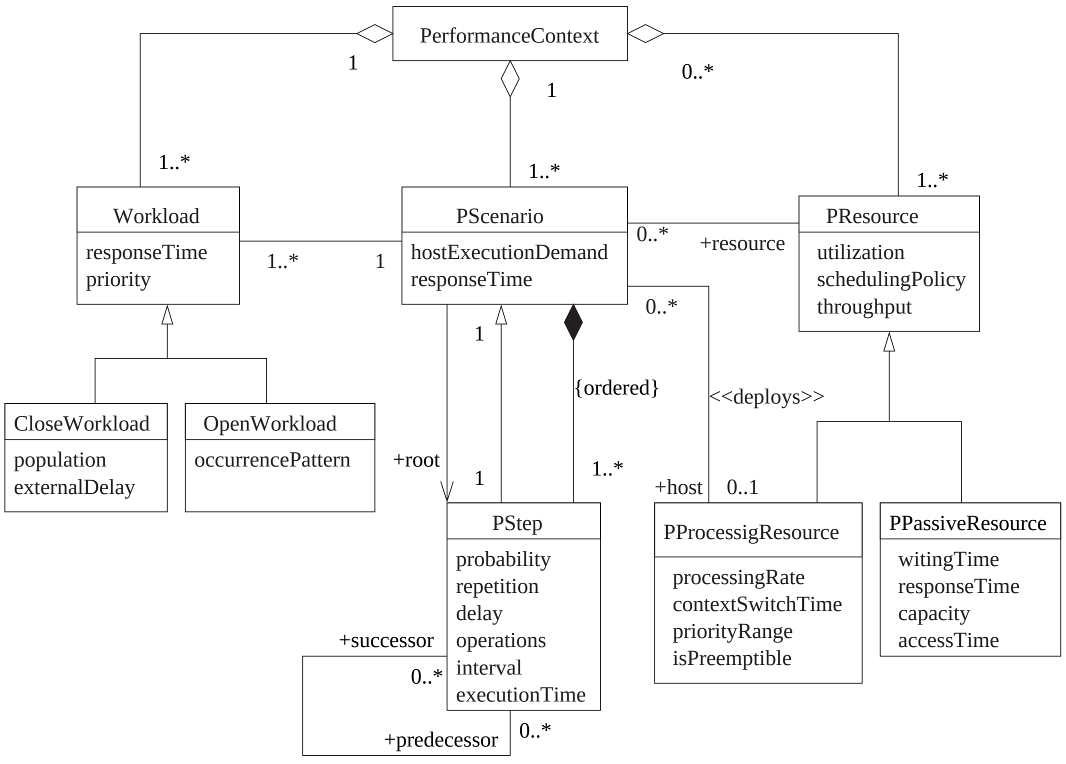


Figure 4.3: The Performance analysis domain model [13]

for varying loads on the system, e.g., estimating the impact of periods of heavy load on the system.

A *performance context* specifies one or more scenarios that are used to investigate various situations of changing loads. Therefore, a performance context also involves a set of resources that are used by the scenarios and a set of workloads applied to the scenarios.

Scenario

As mentioned, a scenario consists of a sequence of scenario steps joined together in a predecessor-successor relationship. The steps of a scenario are executed on (potentially different) host resources. A host resource is only specified for a scenario if all the steps constituting the scenario execute on the *same* host resource. Different workloads can be applied to the scenario. The scenario element has two attributes: The *hostExecutionDemand* attribute denotes the total execution demand (service requirement) of the scenario if all the steps execute on the same host. Otherwise it is not specified. The *responseTime* attribute states the total time required to execute the scenario.

Step

A step indicates an increment in the execution of a scenario and a step is related to other steps in the aforementioned predecessor-successor relationship. A step is specific to a certain scenario and in general a step takes finite time to execute. A step may describe an operation of any level of granularity appropriate for the current modelling effort. Later, the step may be decomposed into finer-grained steps. Consequently, scenario steps are modelled as a sub-type of scenarios which allows for a hierarchical decomposition of scenarios. A step has the following attributes:

- *hostExecutionDemand* (inherited from Scenario). If a step is defined at the finest granularity it executes on a unique host resource and this is the total execution demand of the step on the resource. Otherwise, if the step has a decomposition into a sub-scenario, this is the total demand of the sub-scenario, assuming that all the steps constituting the sub-scenario execute on the same host.
- *delay*. This is an inserted delay (e.g., think time) within the step.
- *probability*. In cases where the predecessor of a step has multiple successors, this is the probability of this step being executed.
- *interval*. If a step is repeated within a scenario, this is the interval between repetitions of the step.
- *repetition*. If a step is repeated within a scenario, this is the number of times the step is repeated.
- *operations*. This specifies operation on resources that are not explicitly represented in the model. Instead, these resources are resolved by the appropriate performance analysis modelling tool.

Resource

This is an abstract view of a resource which may be either *passive* or *active*. We will discuss active and passive resources in the following. A resource may participate (be utilized) in one or more scenarios of a performance context. A resource has the following attributes:

- *utilization*. The value of this attribute is often supplied as the result of model analysis (e.g., solving a QNM) and indicates the average percent of the time that the resource is being utilized.
- *throughput*. This is the average rate at which the resource perform its function.
- *schedulingPolicy*. This is the scheduling policy in effect for accessing the resource.

ProcessingResource

A processing resource is a device such as a processor, interface device or storage device. A processingResource has the following attributes:

- *processingRate*, which is the relative speed factor for the processor.
- *contextSwithTime*. This is the amount of time it takes for the processing resouce to switch from the execution of one scenario to a different one.
- *priorityRange*. This is the permissible range of priorities with which resource actions can be executed.
- *isPreemptible*. This is a boolean value that indicates if the processor is preemptible once it begins execution of an action.

Passive Resouce

A passive resource is a resource that is protected by an access mechanism. Concurrent access to the resource is restricted according to some access control policy. The following attributes characterizes a passive resource:

- *capacity*. This is a measure of the maximum amount of concurrent users of the resource.
- *accessTime*. This is the delay period a scenario incurs when acquiring and releasing the resource.
- *utilization*. Utilization for a passive resource is expressed as the average number of concurrent users of the resource.
- *responseTime*. This is the total amount of time elapsed from the point of acquiring the resource until the point of releasing the resource.

- *waitingTime*. This is the elapsed time from a resource request until the request is granted.

Workload

Workload is an abstraction that specifies the execution demand for a a given scenario. In addition, required or estimated response time for the scenario pertaining to this workload is given. A workload can be either open or closed. The attributes of a workload are:

- *responseTime*. The total amount of time from the corresponding scenario is started until it is completed.
- *priority*. This is the priority of the workload.

OpenWorkload

As mentioned, an open workload is characterized by requests arriving continuously into the system at a fixed rate. An openWorkload has a single attribute:

- *occurencePattern*. This denotes the interval between arriving requests.

ClosedWorkload

In a closed workload, a fixed number of jobs (or users) cycle between executing the scenario and spending an external think time delay. The attributes are:

- *population*. This is the size of the workload (number of users/jobs)
- *externalDelay*. This is the think time delay, i.e., the delay between the end of one response and the start of the next.

The next section describes how the concepts of the performance domain model are mapped to extensions to the UML metamodel.

4.2.2 Mapping the domain model

Scenarios can be modelled as either activity- or collaboration diagrams. In our approach, we will be using activity diagrams to model scenarios, so we will concentrate the discussion on mapping the performance concepts onto UML extensions to an activity diagram. As mentioned, the extensions to the UML metamodel is expressed via stereotypes, tagged value and constraints.

Using activity graphs, as opposed to using collaboration diagrams, gives the advantage that a scenario can be decomposed into a hierarchical structure, that is, an activity graph may consist of several lower-level activity graphs.

PerformanceContext

A performance context can be modelled by an activity diagram stereotyped by the «PAcontext»stereotype. (All extension element names of the performance portion of the UML-SPT profile are prefixed with "PA".) We will defer a discussion of the details of the extension elements, but instead refer to [13] for further details. In case of a scenario containing lower-level activity graphs, only the performance context of the topmost level can have a workload defined. A requirement concerning the activity diagram is that the swimlanes can only be used to represent the resources used or the object instances participating in the activity.

Scenario

Scenarios are modelled by the activities (also called states or actions) and transition between activities of the activity graph. The UML-SPT profile does not define an explicit scenario stereotype, but instead identifies the scenario by the first step (root step) of the scenario. Also, workload information is attached to the root step.

Step

Each activity or sub-activity of the activity graph is stereotyped as a «PAstep». Workload information can be attached to a step if it is the root step of the topmost performance context. This workload is then imposed on steps of lower-level performance contexts.

Workload

The workload for a scenario is modelled by a tagged value associated with the root step of the topmost performance context. Optionally, the root step can be stereotyped as a «PAOpenLoad»or «PAClosedLoad».

ProcessingResouce

A processing resource can be modelled in two ways. Either by associating the appropriate stereotype «PAhost»with a partition (swimlane). This approach, however, is only appropriate in situations where where each instance is executing on its own host. Alternatively, in cases where swimlanes represent instances executing on different hosts, and some instances may share hosts the activity diagram can be coupled with a deployment diagram. This is the most common situation.

PassiveResource

Passive resources are represented by swimlanes stereotyped with the «PAresource»stereotype.

Part II

Our Approach

5 Describing our approach

Our approach, in this thesis, is targeted at performance on the J2ME platform. The motivation for choosing the J2ME platform is that device characteristics are very limited and applications are thus particularly susceptible to performance degradations. Two areas in particular present a challenge on the J2ME platform: network usage and persistence. There are widely differing performance characteristics between local and remote operations and similarly between operations using persistence and operations that do not. These effects are further compounded on the J2ME platform. Therefore, these two areas are the focus in this thesis.

5.1 The Record Management System

Persistence in J2ME is implemented via the Record Management System (RMS), an API that provides applications with local, on-device persistence. As the name implies, data are saved into records, which can be a somewhat misleading term since records does not contain any fields. Rather, data are saved into a record as an array of bytes. It is not possible to change selected parts of a record - instead an entire record must be read, changes to the data must be made in memory, and finally the entire record must be written back [12].

A record can be used to store the state of a class (i.e., the value of its fields) persistently by converting the fields of the class into a byte array which is inserted into a record.

Records are logically collected in to record stores. In order to work on a record store it must be first be opened. Correspondingly, an open record store must at some point be closed.

5.2 Networking

Use of networking on a mobile device can prove problematic due to the limited network bandwidth. There are a few different way a mobile device can access the internet. The most common ways is through either GSM-CSD, which operates at 9,6 kbps or GSM-GPRS which has a theoretical maximum of 172 kbps. Furthermore standards like EDGE which operates at speed upto 55 kbps and HSCSD at 28.8 kbps also exist. The new 3G standard which is becoming increasingly more widespread operates at speeds upto 384 kbps depending on how you are positioned in relation to the antenna. These speeds does however not match the speed of wired networks which operates at speeds between 512 kbps and several mbps for the typical ADSL connection and between 10 mbps and 1gbps for local area networks.

In this installment we are using web services. Web services are supported by the JSR-172 API in J2ME. However at the present time the JSR-172 API has only been adopted by very few devices - as an example Nokia only has developed three phones so far supporting the JSR-172 API, the N70, N90, and N91 [1–3] and these are still rather expensive. As a consequence, in order to communicate with web services on mobile devices third party libraries must be used. In this thesis we have focused on the use of kSOAP 1.2 [16, 17] as the web service library used to communicate with web services.

5.3 Obtaining input estimates

We have created a test application which tests the performance characteristics of the RMS and networking.
The test application can be used by a developer to obtain input estimates which are used to annotate the activity diagrams.

5.3.1 Testing the RMS

The RMS part of the test application allows the developer to specify the number of record stores, the number of records in each record store and the size of each record.

Then the test application tests the time it takes to open and close record stores, read from and write to records, and the time it takes to delete the record stores.

The number of record stores to be input depend on the number of container classes that are to be saved persistently (e.g., customers, visits).

The number of records in each record store depend on the number of objects in the container classes (e.g., the number of customers).

Finally, the last parameter, record size, can be determined by investigating the fields of the objects to be saved. The size of a record can be determined by summing the size of the Java primitive data types of all the fields of the object. Table 5.1 shows the size of the data types which are defined by the Java Language Specification [48].

data type	size
byte	1 byte
char	2 bytes
short	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

Table 5.1: Size of Java primitives

To illustrate, we will use a simplified customer class, which is shown in listing 5.1.

```
public class Customer {
    private int custNum;
    private String name;
}
```

Listing 5.1: Example customer class

When saving a string into a record (i.e., converting it into a byte array) the resulting size of the byte array is the number of characters combined with a delimiter in each end. However, since the final size of a string is not known before it is implemented we have opted for setting a fixed serialized string size at 150 bytes, which corresponds to a string of length 148 characters. The choice of 150 bytes is that we feel that this size will suffice in the majority of cases and at the same time this is not an artificially high number.

In this way, the record size for saving the class customer will be: 4 (int) + 150 (String) = 154 bytes.

Some UML tools, such as IBM's Rational XDE Developer [19], are able to derive and present the size of classes based on a class diagram.

5.3.2 Testing the network

On a wireless device the process of requesting data from a web service using kSOAP consists of these four parts:

1. Creating a SOAP message to send to the web service.
2. Calling the webservice with the SOAP message just created.
3. Getting the result from the webservice.
4. Converting the received SOAP message into an appropriate data structure to be used in the application.

The test application measures the total amount of time it takes from the point of creating the SOAP message until the received SOAP message is converted into an appropriate data structure. The reason for this approach is that it allows the developer to performance model the application without any knowledge about how the SOAP message is generated, sent, parsed, and converted into a data structures.

As the overall load of the network, both data and speak, has an effect on the bandwidth available at any given time, it is imperative that the network test be conducted at a time of the day that is representative of the users "usage pattern". Obviously, conducting tests at a time that does not correlate to the actual use of end users produces results that may give a false indication as to the performance of network operation times. For instance, the load between 8 a.m. and 9 a.m., when a lot of people are just starting their workday, can be expected to be greater than the load between 2 a.m. and 3 a.m.

5.4 Using the results

The results obtained from running the test application are used to annotate the activity diagrams describing the scenarios of a system with respect to operations on the RMS and network. Then the performance annotated diagrams are used to derive performance estimates of the system based on the algorithm for computing Execution Graphs(EG). EGs will suffice since their is no contention for resources on a J2ME device, and the web service that is contacted is treated as a black-box.

We deliver the results of the tests as three statistical properties: The average, maximum and 95th percentile [47] response times. The reason for including the 95th percentile is that it is generally less sensitive to odd spikes in the response time than the average value can be. This is however only true if the data set is sufficiently large.

6 Applying our approach

In this chapter we will give an example of how the input estimates from the test application are used to annotate the activity diagrams describing the scenarios of the system. We will also illustrate how the input estimates are used as a basis for computing the performance predictions. As a test case we will be using a system that we developed as part of a previous project in the fall of 2004 [28].

6.1 A description of the test case

The test case is an end-to-end system consisting of a J2ME application communicating with a web service. It is a package registration system developed to aid a logistics company in keeping track of their package elements. The logistics company functions as a central storage for a number of wholesale dealers (customers) and the aim of the system is to help optimize the process of registering package elements that are used to transport goods as they move from the logistics company to the wholesale dealers and back again.

The scenario is this:
At the warehouse of the logistics company products that are to be shipped out, are packaged and loaded onto a truck. The truck driver receives a route of where the products are to be delivered. At each customer along the route the truck driver delivers the products (and the package elements used to carry the products) and in return he loads onto the truck empty package elements from the customer which he

returns to the logistics company. The truck driver registers the amount of package elements delivered and received, respectively. At the end of the route the truck driver returns to the logistics company and fills out a paper form of the amount of package elements that was delivered and received at each customer.

The logistics company wanted to streamline this process and dispense with the paper work. The solution was to equip each truck driver with a mobile phone and develop a J2ME application which could be used by the truck driver to enter package information pertaining to a particular route. At the start of the day, the J2ME application allowed the truck driver to fetch his route from a web service and as the route was completed the route could be sent back to the web service.

A requirement to the system was that it would be able to recover gracefully from crashes and information should also be saved persistently if the data could not be sent to the web service.

The architecture of the system is depicted in Figure 6.1.

We will now give an overall view of the functionality of the system by describing the user-application interaction. The user-application interaction is illustrated by a high-level activity diagram i figure 6.2.

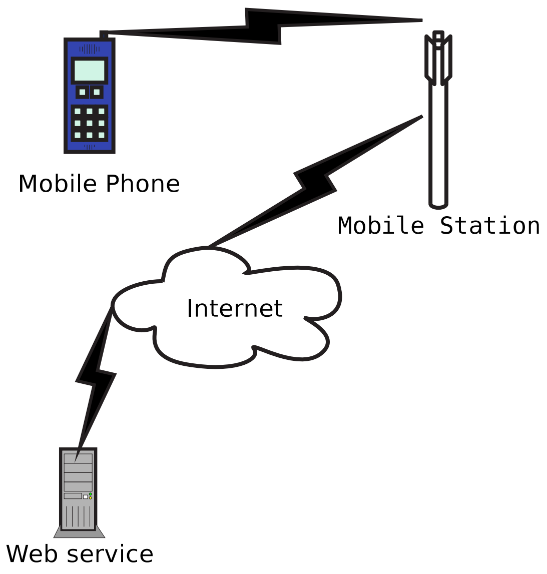


Figure 6.1: Test application architecure

The user starts the application. Upon startup, the application examines if persistent data is saved since the last invocation of the application. This situation occurs if the truck driver has not yet sent data from the last route he worked on back to the

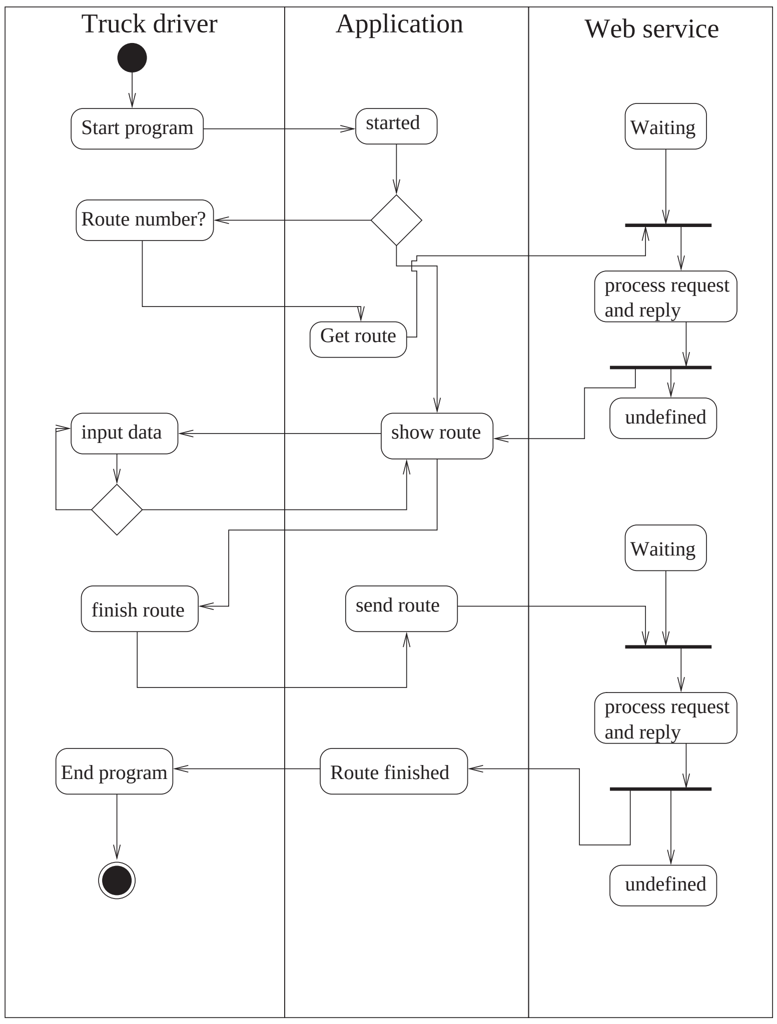


Figure 6.2: The overall activity diagram

web service. This may be facilitated by a conscious choice by the truck driver (i.e., a choice to close the application), a failure to send the data back to the web service, or a power failure or some other condition causing the application to shut down. In this case the application resumes operation by showing an overview of the previous route and the progress of the route. Otherwise, if no persistent data exist, the application queries the user for a route with a particular route number to be fetched. The truck driver plots in a route number and the application sends a request to the web service for the specified route. The route is transmitted to the device, the data are saved persistently, and the user is shown an overview of the route (the collection of customers to be visited). For each customer, the user inputs the appropriate data, which happens through a series of input screens. Upon completion of a route, the truck driver chooses “finish route” and the data are sent back to the web service. If the data was successfully transmitted, the persistent storage (RMS) is “wiped clean”, otherwise the data remain in the RMS until a successful transmission can be completed.

The key performance scenarios selected for performance annotations are: “finish route”, “get route”, “‘read persistent” and “save persistent” since these are the ones involving the network and persistent storage.

6.2 Scenarios

In the following the selected key performance scenarios are described in activity diagrams. These activity diagrams are annotated, as described in Chapter 4, with performance information obtained through experiments with the test application.

It should be noted that the performance annotations attached to the activity diagrams are derived from a single run of the test application. As it is, we have conducted several tests of both the test application as well as the logistics applications in order to account for possible variance in the results.

6.2.1 Get route

The activity diagram for the following scenario can be seen in Figure 6.3. The truck driver starts the application and the application checks for persistent data indicating that the previous route is unfinished. If this is the case we proceed to the “read persistent data” sub-activity diagram. The performance annotations attached to the “read persistent data” step denotes the accumulated response time for the sub-steps (expressed in the sub-activity diagram). We can see that the execution time (or host execution demand), PAdemand, for the step is 361.2 milliseconds on average (’mean’) and this is a measured (’msr’) value (by our test application).

If no persistent data exist the application starts afresh and the truck driver is queried for the route number. Assuming a valid route number, the route is fetched from the web service. As can be seen from the performance annotation on the “Fetch route” step the time it takes to fetch a route has been measured at an average of 23974.8 milliseconds by our test application (or rather the time it takes to fetch 36189 bytes of data).

Upon successful reception of a route from the web service, the data are saved persistently. This process is specified in the "save persistent” sub-activity diagram and the accumulated measured mean time is correspondingly attached to the “save persistent” state.

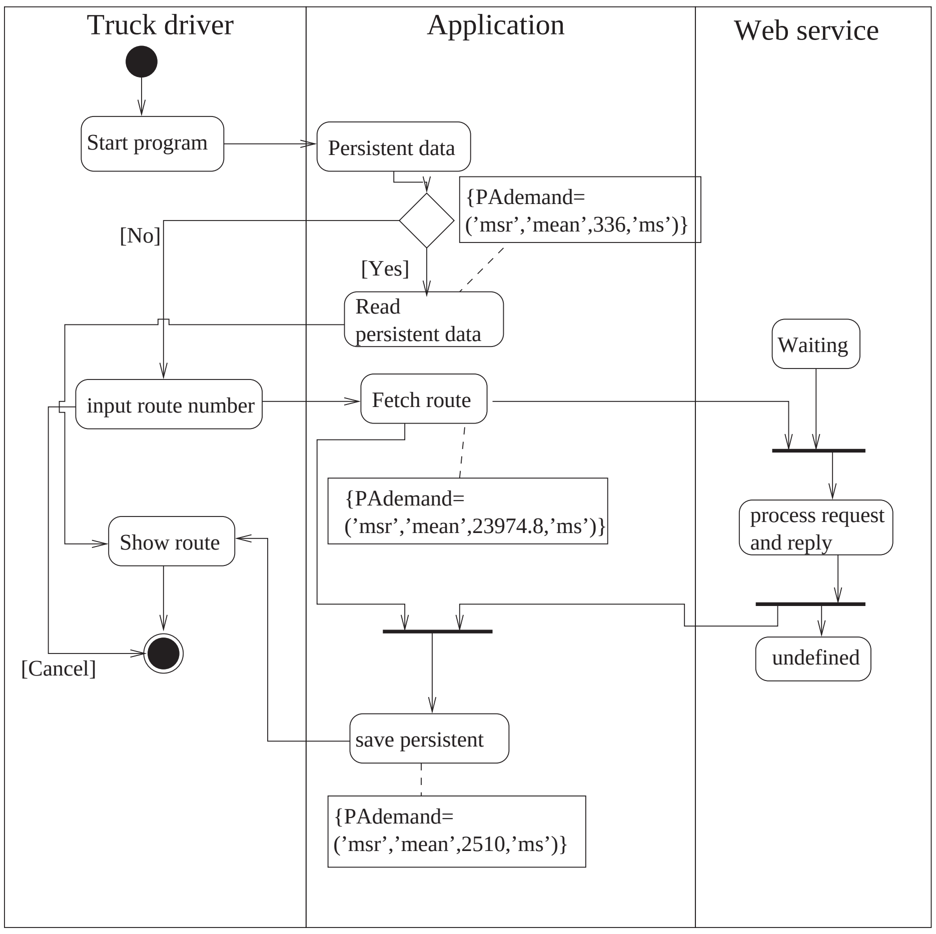


Figure 6.3: Get Route w/time activity diagram

As described, the “get route” activity can take two directions pending on whether or not persistent data are present - as modelled by the forking at the “persistent data?” state. In the case of persistent data being present the total time it takes before the route can be shown to the user is determined by the execution demand of the “read persistent data” state which is estimated to be 361.2 milliseconds by our test application. Conversely, if no persistent data exist, the total time it takes before a route can be shown is estimated to be 23974.8 milliseconds ("Fetch route") followed by 2529.6 milliseconds incurred by saving the route persistently totalling 26504.4 milliseconds.

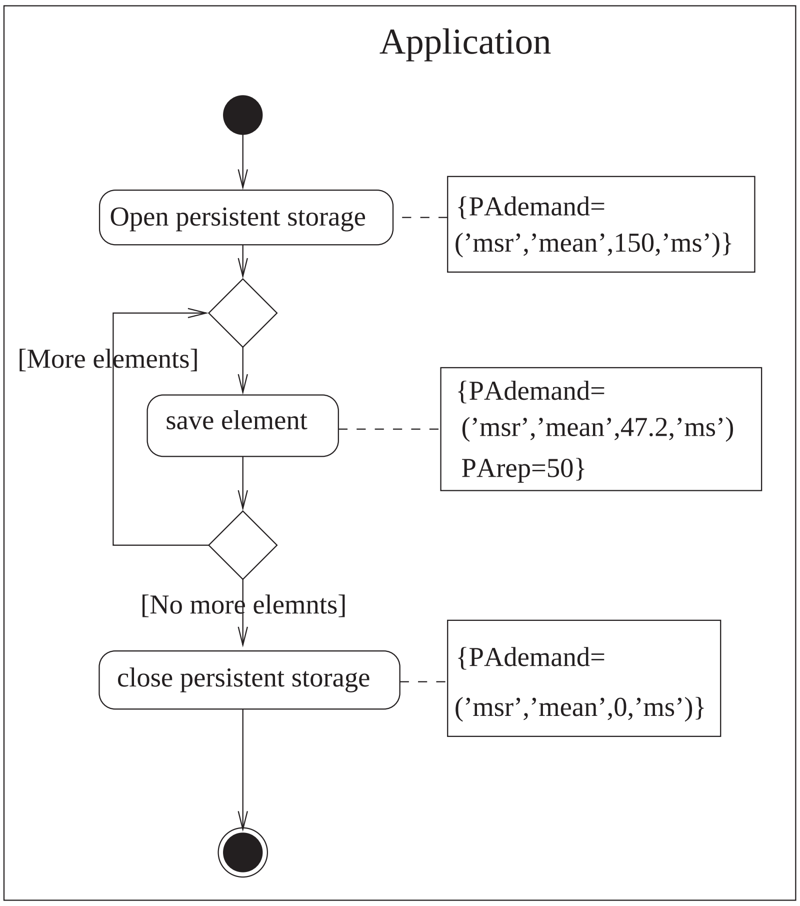

Application
Open persistent storage
{PAdemand=('msr','mean',150,'ms')}
[More elements]
save element
{PAdemand=('msr','mean',47.2,'ms') PArep=50}
[No more elemnts]
close persistent storage
{PAdemand=('msr','mean',0,'ms')}

Figure 6.4: Save persistent w/time activity diagram

6.2.2 Save persistent

Saving data in most cases is a three step process consisting of the steps of opening a record store, saving records to the record store, and closing the record store upon completion. Alternatively, a static reference to an open record store can be kept in which case the opening and closing of a record store are omitted from this process. As can be seen from the activity diagram in figure 6.4 we are modelling the first case. The average measured time to open a record store is derived from the results of running our test application and the "open persistent storage" step is annotated with this information.

The average time to write a record (save an element) is measured at 47.2 milliseconds and the number of elements to be saved in this example is 50 which is indicated by setting PArep (the repetition factor) to 50.

Thirdly, the appropriate average measured value is attached to the "close persistent storage" step.

These three steps happen in succession and the total execution time for writing 50 records (including opening and closing the record store) is thus 150 + (47.2 * 50) + 0 = 2510 which, as we saw in the "get route" activity diagram, is the value given to the state encompassing this sub-activity.

6.2.3 Read persistent

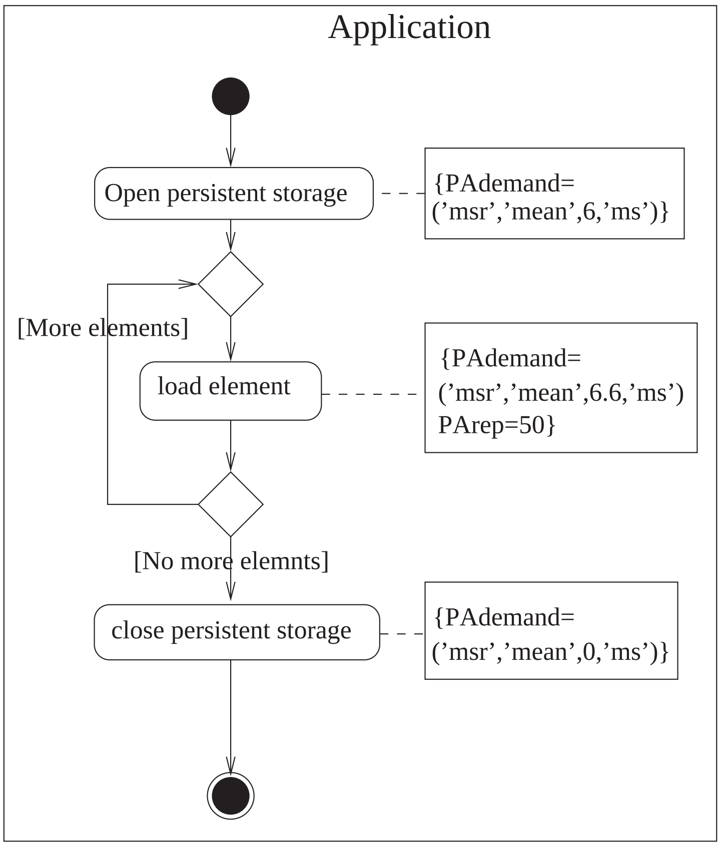


Figure 6.5: Read persistent w/time activity diagram

Reading data from the RMS, illustrated in Figure 6.5, follow a path similar to that of writing to the RMS. More specifically, it consists of the steps of opening a record store, performing the reads (i.e., iterating over the record store and reading the records) and closing the record store. As before, the steps are annotated with the appropriate performance information obtained from running the test application and the resulting execution time for reading 50 elements from the RMS in this case adds to 6 + (6.6 * 50) + 0 = 336 milliseconds which is given as the host execution demand for the "read persistent" step.

6.2.4 Abruption

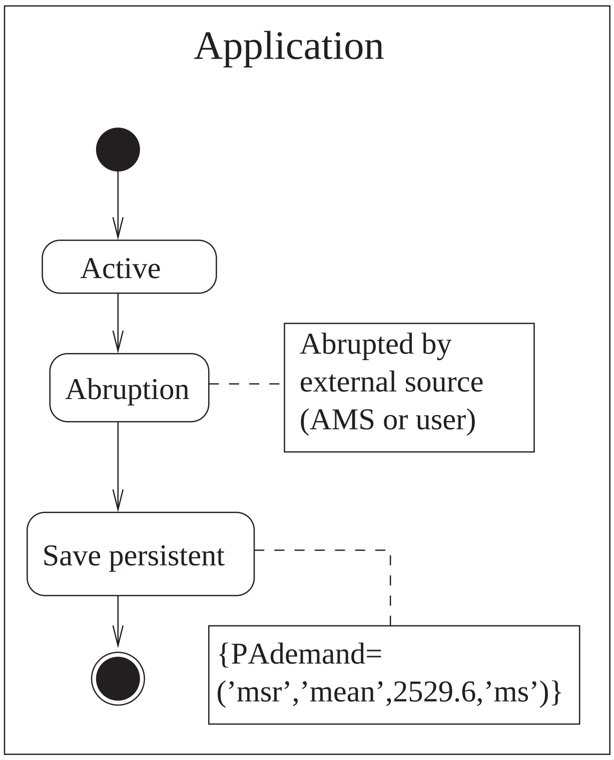

Application
Active
Abruption
Abrupted by external source (AMS or user)
Save persistent
{PAdemand= ('msr','mean',2529.6,'ms')}

Figure 6.6: Abruption w/with activity diagram

To account for a sudden abruption of application execution, data must be saved persistently in order to ensure that the present route can be resumed on the next startup. The procedure at this point is identical to saving the data when the route was received from the web service as described in 6.2.2

6.2.5 Finish route

When the truck driver chooses "finish route" this initiates a send to the web service. The average estimated time it takes to send 75574 bytes of data, as reported by the test application, corresponding to the size of the constructed SOAP message of the route to be sent is used to annotate the "Send Route" step.

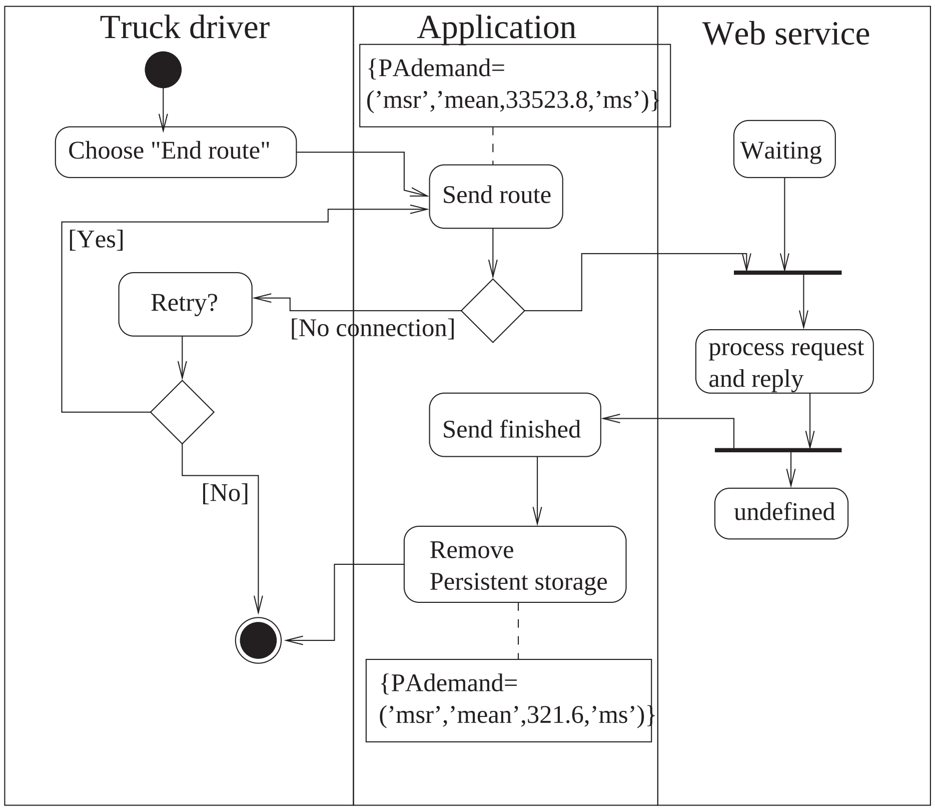


Figure 6.7: Finish route w/ time activity diagram

When the application receives an acknowledgement from the web wervice indicating that the transmission was successful, the RMS is deleted.

The estimated average value for deleting a record store, as reported by the test application, is annotated to the "delete persistent storage" step. The estimated total execution time for a successful send then comes to 33523.8(for "Send Route") + 321.6 (for "delete persistent storage") totalling 33845.4 milliseconds.

7 Evaluating our approach

In this chapter we investigate the extent to which our objective of creating an approach that is useable, reliable, transparent, and cost-efficient are met. The bulk of the chapter is devoted to an investigation on the accuracy of the input estimates supplied by the test application.

We present, compare and investigate the results obtained by running the test application and test case (logistics application) ten times each. The tests have been conducted ten times each in order to account for variance in the results. The results we use for comparison are taken as the average of the ten tests. For the logistics application we have measured the best-case, worst-case, and average case scenarios. For instance, this means that the worst-case value we will use is the value obtained by taking the average value of all the worst-case values and similarly for the the best- and average cases.

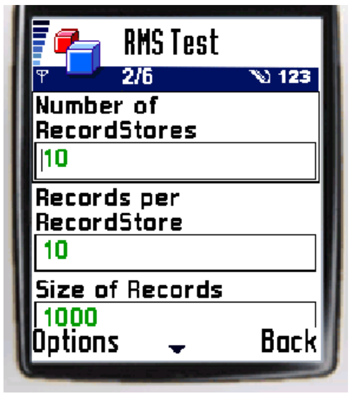
The results from running the test application are delivered as three statistical measures: the average, maximum and 95th percentile results. These are used to measure the times of *single* operations (e.g., the time it takes to open a single record store or the time it takes to read a single record from a record store). In addition, the test application also return the total elapsed time for reading and writing all the elements of a record store. As with the results for the logistics application we use the average value obtained over the course of the ten runs. In essence, we take the average of the ten maximum times, average times, and 95th percentile times.

According to Daniel A. Menascé, ACM Fellow, an error margin of up to 30% is considered acceptable in performance analysis [40]. We will investigate whether this aim is achieved by the performance results returned from our test application.

7.1 Evaluating the RMS results

We have used the test application to emulate a route consisting of 50 visits to 50 customers combined into a single record store. Originally these two containers were placed into separate record stores (customers and visits) in the logistics application but due to the design of our test application we have decided to collapse the two containers into a single record store.

The reason for this is that our test application in its current version only allows a single parameter input to be specified for the number of records and the size of records (the input screen for the RMS test can be seen in Figure 7.1). Obviously, the number of records in different record stores as well the size of these records would probably vary in a real-world application. This design choice in our test application was made for reasons of speedy development since we are focusing on investigating the accuracy of the input estimates supplied by our test application rather than concentrating on making the test application user friendly in this first version. We thus found it faster to collapse the two record stores of the the logistics application into a single record store. It is of course still possible to emulate different record stores with different number of records and record sizes with our test application but this requires the test application to be executed separately for each record store. However, splitting up the tests this way will not give a representative workload emulation and possible memory problems might escape undiscovered. It should be noted that the appropriate changes to the test application can easily be incorporated into a second version of the test application.



Number of RecordStores: 10
Records per RecordStore: 10
Size of Records: 1000

Figure 7.1: The input screen for the RMS test on the test application

So, to emulate the logistics application we have conducted ten tests on the test application involving a single record store with 50 records containing the combined data of customers and visits. The size of the records come to a total of 637 bytes ((4 $\times$ 150 for Strings) + (9 $\times$ 4 for ints) + (1 for boolean) according to our assumption of serialized strings occupying 150 bytes. The class diagram for a route can be seen in Figure 7.1.

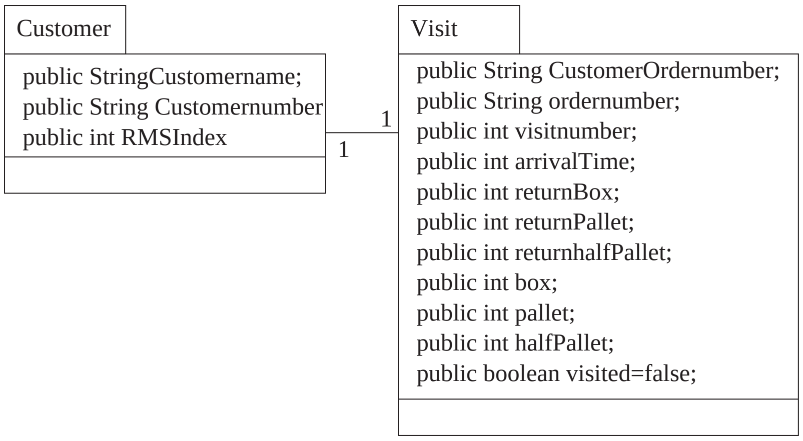


Figure 7.2: The class diagram for the route

Table 7.1 shows the results of the logistics application specified as the average of the average, minimum (best case) and maximum (worst case) total times measured in milliseconds for reading, writing and deleting 50 records over the course of ten runs.

Test	avg. total time	max. total time	min. total time
Reading	240,5 ms.	406 ms.	186 ms.
Writing	3671,8 ms.	3812 ms.	3548 ms.
Deleting	999,8 ms.	1047 ms.	953 ms.

Table 7.1: Results of RMS operations on the logistics application

Table 7.2 shows the results of our test application emulating the logistics application on ten different runs.

Test	avg.	max	95th percentile	total
Create	126,7 ms.	234 ms.	141 ms.	–
Open	3 ms.	15 ms.	15 ms.	–
Close	0 ms.	0 ms.	0 ms.	–
Read	6,9 ms.	50,2 ms.	16 ms.	372,1 ms.
Write	47,7 ms.	135,9 ms.	96,8 ms.	2481,3 ms.
Delete	323,3 ms.	344 ms.	343 ms.	–

Table 7.2: The results returned from the test application emulating the workload of the logistics application

It should be noted that for the 95th percentile value given for the operations that happen only once in each test (i.e., create, open, close, and delete record store) we have somewhat artificially supplied the second highest value of the ten runs as the 95th percentile. This is because in order to ensure that the 95th percentile is not (guaranteed to be) equal to the maximum value, the data set (in this case number of tests) must be at least 20.

The only exception to the "individual operation times" is the "total" column which gives the total elapsed time for reading 50 records and writing 50 records.

7.1.1 Comparing the results

Comparing the write results

In order to compare the results we should note that the write times on the logistics application encompass a "create record store" operation and a "close record store operation". That is, when data are written persistently on the logistics application, a new record store is created and after the writes are performed the new record store is closed. Accordingly, the test results from the test application for writing must also include the test times for the create record store and close record store operation.

The subset of results from the test application that are relevant for write operations are shown in Table 7.3. The second, third, and fourth entry in the "Write" row show the average, maximum, and 95th percentile results for writing a single record, whereas the fifth column shows the total execution time for saving all 50 records as measured by the test application.

Test	avg.	max	95th percentile	total
Create	126,7 ms.	234 ms.	141 ms.	–
Write	47,7 ms.	135,9 ms.	96,8 ms.	2481,3 ms.
Close	0 ms.	0 ms.	0 ms.	–

Table 7.3: Results of operations relevant for performing writes obtained from the test application

In order to more directly observe the difference between the total write time on the logistics application and the total write time on the test application, both the one measured directly and the ones computed using the different measures we perform the calculations "create record store time" + (50 $\times$ write record time) + "close record store time" for the average, maximum and 95th percentile results. The resulting estimates can be seen in Table 7.4

Test	total execution time
total (measured)	2608 ms.
avg. (50 times)	2511,7 ms.
95th percentile (50 times)	4981 ms.
max. (50 times)	7029 ms.

Table 7.4: The total write time computed for the avg., max., and 95th measures as well as the total time measured

Test	avg. total time	max. total time	min. total time
Writing	3671,8 ms.	3812 ms.	3548 ms.

Table 7.5: The average, minimum, and maximum times for writing on the logistics application

We contrast these results to the write results of the logistics application (seen in Table 7.5)

In order to see how the estimates of the test application are positioned with respect to the actual write times of the logistics application we have shown the percent-wise deviation of the test application's estimates to the logistic applications actual measurements in Table 7.6.

Test app./ Logistics app.	avg.	max.	min
total	-28,97%	-31,58%	-26,49%
avg.	-31,59%	-34,11%	-29,21%
max.	+91,43%	+84,39%	+98,11%
95th percentile	+35,66%	+30,67%	+40,39%

Table 7.6: The percent-wise deviation between the estimates of the test application and the actual results of the logistics application for write operations.

The results in Table 7.6 has been prefixed with + and -, indicating whether the test application provides an overestimate or underestimate, respectively, of the logistics application's performance.
From Table 7.6 we can see that the measured total time provides us with the most accurate estimates scoring within the 30% range on two occasions (with a deviation at 28,97% of the logistics application's average case and 26,49% of the logistics application's best case time). In the case of estimating the worst case, the measured total time scores just outside at 31.58%. Only one of the single-operation measures is able to achieve within the 30% error margin, the average measure at a 29,21 deviation of the logistics application's best case write time while scoring just outside the range of the logistic application's average case and somewhat close to the 30% range of the worst case.

We can see that the maximum estimated time from the test application is of little usage in estimating the performance of the logistics application. This is not surprising, as this is the estimate arrived at by taking the single longest write time and compounding the effect of this by assuming that all records take this time to complete. We have observed quite a significant difference between the average case and the occasional spike in write time incurred by the garbage collector (presumably) so this result could be expected. The reason for including it was to provide a worst case estimate.

The 95th percentile measure scores slightly worse than the average measure (in two out of three cases). Only in the case of estimating the worst case performance of the logistics application is the 95th percentile more accurate. Our primary concern in general are the average and worst case performance that we can expect to achieve from an application, and in the latter case the 95th percentile measure yields the more accurate estimate, just outside the 30% range (at 30,67%) thus giving the best worst case estimate. An added benefit of the 95th percentile is that it errs on the right side of caution so to speak, in that it does not give underestimates of the actual performance.

Comparing the read results

As for writes, reads also incur two additional operations. In this case an "open record store" operation as well as a "close record store" operation.

We show the results of the read-relevant operations from the test application in Table 7.7

Test	avg.	max	95th percentile	total
Open	3 ms.	15 ms.	15 ms.	–
Read	6,9 ms.	50,2 ms.	16 ms.	372,1 ms.
Close	0 ms.	0 ms.	0 ms.	–

Table 7.7: Results of operations relevant for performing reads obtained from the test application

Also, to get a clearer picture of the differences between the estimates supplied by using the different measures we give their resulting estimates ("time of open record store operation" + "50 $\times$ "individual read operations" + "close record store operation". These results can be seen in Table 7.8

We recap the read times from the logistics application in Table 7.9

We then show the accuracy of the read predictions from the test application in relation to their percent-wise deviation from the actual read results measured on

Test	total execution time
total (measured)	375,1 ms.
avg. (50 times)	348 ms.
95th percentile (50 times)	815 ms.
max. (50 times)	2525 ms.

Table 7.8: The total read time computed for the avg., max., and 95th measures as well as the total read time measured

Test	avg. total time	max. total time	min. total time
Reading	240,5 ms.	406 ms.	186 ms.

Table 7.9: The average, minimum, and maximum times for reading on the logistics application

the logistics application. This can be seen in Table 7.10.

Test app./ Logistics app.	avg.	max.	min
total	+55,97%	-7,61%	+101,67%
avg.	+44,52%	-14,29%	+87,09%
max.	+949,89%	+521,92%	+1257,53%
95th percentile	+238,88%	+100,74%	+338,17%

Table 7.10: The percent-wise deviation between the estimates of the test application and the actual results of the logistics application for read operations

As table 7.10 clearly shows the estimates provided by the test application are rather fragmented. Only in two cases are the estimates within the 30% range of the results of the target application. The total measured value from the test application comes close to the worst case read time of the logistics application at a 7,7% deviation while being more than 100% inaccurate in predicting the best case time. These results are of course caused by the very large difference between the best- and worst case read times on the logistics application.

Overall, the total measured read time and average read measure provide the best estimates in this test positioning themselves between the average and worst case time on the logistics application and with both being within range of the worst case time, at a 7.61% and 14,29% deviation respectively.

Both the 95th percentile and maximum estimates from the test application far exceed the actual read performance of the logistics application.

Comparing the delete results

In contrast to the reads and writes discussed in the previous sections, deleting a record store does not include additional operations. Hence, we can directly observed the delete times.

The estimated times for deleting as reported by the test application are shown in Table 7.11

Test	avg.	max	95th percentile
Delete	323,3 ms.	344 ms.	343 ms.

Table 7.11: Results of delete operations obtained from the test application

In relation to the delete times measured on the logistics application and seen in Table 7.12 the percent-wise deviations of the delete times obtained by the test application's are shown in Table 7.13

Test	avg. total time	max. total time	min. total time
Deleting	999,8 ms.	1047 ms.	953 ms.

Table 7.12: The average, minimum, and maximum times for deleting a record store on the logistics application

Test app./ Logistics app.	avg.	max.	min
avg.	-209,25%	-223,85%	-194,78%
max	-190,64%	-204,36%	-177,03%
95th percentile	-191,49%	205,25%	-177,84%

Table 7.13: The percent-wise deviation between the estimates of the test application and the actual results of the logistics application for delete operations

As the deviations in Table 7.13 clearly shows, the performance estimates from the test application in this test will not suffice in providing (even approximately) accurate estimates.

7.2 Conlusion on the RMS test results

The results of the RMS tests indicate that the total time measured and the average value of single operations multiplied with the number of records provide the most accurate estimates of the target platform's performance. However, as the delete test illustrated these are not always accurate enough. We suspect, that the reason for the large difference in these results is that the delete test is conducted on the logistics application at a time when the garbage collector is reclaiming objects whereas the test application conducts a "clean" RMS test meaning no additional objects are created and destroyed. The garbage collector may be the cause of the anomalies in the estimates. One way to investigate this would be to force a garbage collection on the target application (for validation purposes) and see whether the performance estimates of the test application and the performance of the logistics application would then be approximately similar. (However, as we were sharing the Nokia phone with another project group also working on their master thesis we were not able to try out this possibility in time.) If indeed the garbage collector proved to be the cause of the anomaly it would be beneficial to try and apply our test application to additional target applications of varying complexities in order to investigate this effect.

Performance on the target application may also vary significantly between individual runs, as we saw from the results in the read test. This, of course, makes it difficult to provide accurate estimates in every single run and also illustrate the general problem of providing accurate performance predictions of object-oriented systems.

7.3 Evaluating the network results

We have conducted five runs of the test application where we emulated fetching 36189 bytes data which contains information about which customers the driver shall visit on his route. We also emulated sending the route back when the day is over and the size of this data is 75574 bytes. When sending back the route more information is sent, e.g. what packaging items have been dropped of and what packaging items has been returned, which is the cause of the larger byte size to send than receive.

Test	avg. time	max time	min time
Get route	28549,8	29297	28141
Send route	39240,2	39953	38484

Table 7.14: Results of Networking operations in the logistics application

The results given in Table 7.14 is for the logistics application. The “get route” times include the time it takes to accept the network connection and to chose a connection profile, and the “send route” include the time it takes to accept the network connection.

The results from the test application are given as the average, maximum, and 95th percentile, and the same issue regarding acceptance of network connection and connection profile also applies to the test application.

Test	avg. time	max time	95th percentile
Get route	23974.8	25000	24109
Send route	33523	36296	35156

Table 7.15: Results of Networking operations in the test application

7.3.1 Comparing the results

In order to compare the results of the network operations we should note that the time at which the tests was conducted was between 10 a.m. and 11 a.m. for the test application and between 9 a.m. and 10 a.m. for the logistics application.

Comparing fetching

Test	avg. time	max time	min time
Get route	28549,8	29297	28141

Table 7.16: Relevant results of Networking operations in the logistics application

Test	avg. time	max time	95th percentile
Get route	23974.8	25000	24109

Table 7.17: Relevant results of Networking operations in the test application

Test app./ Logistics app.	avg.	max.	min
Get route	16,02%	14,67%	14,94%

Table 7.18: The percent-wise deviation between the estimates of the test application and the actual results of the logistics application for fetching operations

We can see from Table 7.18 that the test application gives quite good estimates, approximately a 15% deviation.

Comparing sending

Test	avg. time	max time	min time
Send route	39240,2	39953	38484

Table 7.19: Relevant results of Networking operations in the logistics application

As we can see from tables 7.19 and 7.20 the logistics application is slightly slower than the test application.

Test	avg. time	max time	95th percentile
Send route	33523,8	36296	35156

Table 7.20: Relevant results of Networking operations in the test application

Test app./ Logistics app.	avg.	max.	min.
Send route	14,57%	9,15%	9,56%

Table 7.21: The percent-wise deviation between the estimates of the test application and the actual results of the logistics application for the send operation

From Table 7.21 we get another not surprisingly result as the same applies as to fetching. The remarkable thing about these deviations is that the maximum and minimum is below 10% and the average is at the same level as the fetch deviations.

7.4 Conclusion on the network results

As it is, the results of the network tests are rather artificial as we are sending and receiving exactly the same SOAP messages on both the test application and logistics application!

However, during the course of this project we have learned that it is not possible to create dynamic proxies on the J2ME platform. Hence, it is not possible to make a test application that is able to dynamically test the time it takes to communicate with a web service. This will continue to be a limitation until (and if) dynamic proxies on the J2ME platform become a reality.

7.5 Overall evaluation of our approach

We will now discuss the extent to which the objectives of *usability*, *reliability*, *transparency*, and *cost-efficiency* have been met by our approach as well as which problems still remain.

- *Usability*. The approach is usable in that no performance knowledge is required in order to run the test application and obtain the input estimates used to annotate the activity diagrams. For the RMS test, all the user needs to specify is the number of containers and the amount of elements in each container. The data sizes for the records to be saved can be easily obtained by summing the size of the fields of the appropriate classes. In effect, no special performance knowledge is required to conduct the tests and achieve the input estimates. The annotation of activity diagrams is conducted as part of the design of the system which means that performance evaluation is effectively fused into the design process rather than being a separate activity. By making performance evaluation an intrinsic part of the design process we also ensure that the performance estimates are always accurate reflections of the current state of the design, hereby removing the concern of having to keep a separate performance modeling effort up to par with the evolving design. An added benefit of our test application is that it implicitly conducts a feasibility study of the system to be built - if the test application throws an OutOfMemory exception with the specified parameters, then certainly the target application is very much at risk of doing so also, as the test application is stripped of excess functionality. Developers can then use this information to consider an alternative design.

- *Reliability* of estimates. We have seen that the estimates of our approach are fairly accurate in the majority of instances when using the total measured time and average individual time as measures. However, in the RMS test we observed that in a single test (delete record store) the estimates were not valid predictions. We attributed this effect to the garbage collector reclaiming objects during the deletion of record stores on the logistics application but we will have to conduct additional tests on other applications, varying from very simple applications to more complex applications, to investigate the impact of the garbage collector further.

- *Transparency*. We have not been able to fully achieve the transparency objective in practice. Instead in our current effort we have relied on an ad-hoc approach to performance modelling using the algorithm for transforming sequence diagrams to execution graphs which can also be applied to activity diagrams. The reason that we are saying "not fully" is that the transformation algorithm is straight-forward to apply and as such does not require knowledge of execution graphs and can thus be performed by non-specialists. However it is not optimal from a transparency viewpoint. Ideally, the construction and solution of an appropriate performance model should be completely hidden from the developer. The UML-SPT profile lends itself towards integration with performance tools by the transformation process described in Chapter 4. As a result, we expect performance tools to integrate more closely with UML tools in the future. As the UML-SPT profile is still rather new (it went into final release on January 2005) no UML tools yet support the performance component of the profile. Also, presently there does not seem to exist a performance tool that, without any intervention from a developer, can take an XML representation of an activity diagram, compute a performance prediction, and report it back into an XML representation that could be imported by a UML modelling tool. Thus, achieving full transparency is not yet possible by use of existing UML and performance tools.

- *Cost-efficiency*. Our approach is cheap to realize. It incurs an initial modest learning curve associated with familiarizing developers with the UML-SPT profile (and with the activity diagram-to-execution graph algorithm in our current installment to compute the result estimates). Otherwise, using our approach is as simple (and cheap) as executing the test application to assimilate the input estimates and annotating the input estimates to the activity diagrams.

As we have noted, since dynamic proxies are not a reality on the J2ME platform, our test application at the present time is not applicable to web services. This limitation can only be resolved once (and indeed if) dynamic proxies become available on the J2ME platform.

Part III

Conclusion and future work

8 Conclusion

Performance can be a make or break factor in determining the success of software projects. Examples to this fact follows:

NASA was forced to delay the launch of a satellite for a least eight months due to the Flight Operation Segment (FOS) software in the satellite having unacceptable response times for developing satelitte schedules and poor performance in analyzing satellite status and telemetry data. The cost of the delay has not been determined, but there is certainly a loss of prestige, and members of Congress questioned NASA's ability to manage the program. [41]

Similar problems have been experienced in the B2C market.

> "In 2001 $25 billion were lost due to web performance issues" (Zona Research) [32]

> "48% of online shoppers gave up trying to buy some products online because web pages took too long to load" (Boston Consulting) [32]

And finally, a domestic example: In what has been dubbed the largest scandal in Danish computer software history, The Amanda system - a system to to aid the Danish public servant caseworkers in handling cases faster and more efficiently - failed miserably in a national stress test by showing response times upwards of an hour and a half. Although the system exhibited a multitude of additional problems [10], these response times by themselves clearly made the system useless.

A common misconception is that Moore's Law has rendered performance considerations superfluous. However, this is far from the case as evidenced by the substantial investments being made in performance management. In 1998 the performance management market stood at about 2 billion dollars - by 2005 this number has grown to just over 4.5 billion dollars [32]. This increase clearly shows that waiting 18 months for Moore's law to take effect simply is not an option in a competitive market.

Despite the importance of performance, performance considerations in traditional software engineering have been subject to a fix-it-later attitude although correcting performance at a late stage in the development process can be extremely costly and in some cases impossible. Moreover, targeting performance by means of performance tuning and optimizations may erode other software quality attributes such as the maintainability and flexibility of the system.

Software Performance Engineering (SPE) tries to address this shortcoming by considering performance early in the development process.

So far, SPE has not been incorporated into the mainstream software engineering (SE) discipline which is underlined by the fact that SPE is indeed still a separate discipline.

A fundamental reason for this may be that performance-related subjects are scarcely being taught at the majority of computer science schools - often performance is mentioned as a side note, and often performance is inadequately or not defined. The problem is worsened when it comes to knowledge of SPE - a survey [21] showed that out of 24 highly ranked computer science schools in the United States only two courses mentioned SPE, and one of these even incorrectly implied that SPE was a specialized discipline for database-centered and real-time applications.

This has made the adoption of SPE into a software development effort highly dependant upon specialized staff being available with an appropriate amount of performance skills and experience.

In this thesis we have pointed to a number of deficiencies of using such an approach.

- The reliance on a performance specialist in order to be able to obtain performance estimates of the design of a system limits the usability of SPE in ordinary software development projects.
- The reliability of the performance estimates of the system is dependent on the level of expertise of the performance specialist who supplies the input estimates.
- SPE is costly - it consists of an iterative process of performance model creation and solution combined with validation and verification of the models in order to obtain increasingly accurate reflections of the system's performance.

We proposed an approach to making early performance evaluation significantly more appealing to incorporation into traditional software development by 1) substituting the input estimates supplied by a performance specialist with a test application that would provide these input estimates instead and 2) hiding details of performance model creation and solution to the developer as well as eradicating the need to posses intricate knowledge of the device characteristics of the devices to be developed to.

The focus of our approach in this thesis has been directed at performance on the J2ME platform where we have investigated performance concerns regarding the two areas that are notoriously "heavy hitters": network usage and persistent storage. The motivation is that the effects of these two areas are felt particularly on devices with limited capabilities such as is the case of wireless devices. We have used an end-to-end application (J2ME-J2EE), which we have developed as part of a previous project, as a test case in order to investigate the validity of our approach.

We came to the following conclusions regarding the usability, realibility, transparency and cost-efficiency of our approach:

Usability Our approach is usable to "ordinary" SE developers in that no special performance skills are required.

Reliability of estimates We conducted a number of tests of our test application as well of our test case application in order to determine whether the input estimates provided by the test application could be used as a basis for predicting the actual performance of the test case application. We observed that the input estimates from our test application in the majority of cases were able to provide reasonably accurate reflections of the performance of the test case. However, in a single test we observed an anomali where the performance of the test case where several factors worse than than the actual performance of the test case. Similarly, we noted that the performance of the test case application in the RMS read test varied substantially with the worst case taking more than twice the time of the best case to complete. This contributes to making performance predictions in every single run of the test application reliable. We attributed these variations to the garbage collector.

Transparency Within the time frame of this thesis we have not been able to implement a tool (possibly an add-in to an existing UML tool) that would be able to compute the final performance predictions of a system based on the input estimates from our test application. Also, no UML and performance tools exist that support the model processing process of of UML diagrams. Instead, we have relied on an ad-hoc execution graph solution of the system.

Cost-efficiency The cost of our approach is negligible. It involves the cost to execute the test application, which can be done within a matter of a very few minutes, as well as the cost of annotating the activity diagrams with performance information, which is done as part of the design.

8.1 Shortcomings and future work

The test application at the present time is not applicable to web services. This is due to the fact that the creation of dynamic proxies is not possbile on the J2ME platform. However, instead of waiting for dynamic proxies to become a reality on the J2ME platform, a future work may consider estimating the size of the SOAP messages to be sent by investigating the WSDL file describing the web service. We could then time the operation of sending and receiving the computed amount of bytes. The network time could then be combined with a test of various XML parsers to try and estimate the total time of using web services.

An immediate future work includes further tests to investigate the impact of garbage collection. We could conduct additional tests of more applications, ranging from simple ones, where the number of objects are limited, to more complex applications carrying a larger memory footprint. It would also be beneficial to conduct these tests on different devices that have different garbage collection implementations.

Regarding the achievement of transparency, once the UML-SPT profile becomes incorporated into UML tools the automatization of the model processing process of UML diagrams will become realizable in practice.

A natural extension to the approach suggested in this thesis would be to include more general performance considerations in J2ME applications, in particular investigating the overhead of using object orientation in wireless devices.

Another direction may be to investigate our approach on other platforms, such as standalone computers and enterprise applications where the heterogenity of both hardware and software far exceeds that of the J2ME platform.

A limitation, not only of our approach but of performance engineering in general, is that timeliness is the only performance measure considered. An application's memory consumption can be critical to its success. If the application uses excessive memory the garbage collector, which may be very primitive on many wireless devices, will have to free the allocated memory - a process which may have a significant impact on the overall performance, particularly on a wireless device.

8.2 Summary

Performance, or a lack thereof, can mean the difference between succes or failure in a software project. A lack of satisfactory performance may cause schedule and cost overruns and in some cases project cancellation. In a competitive market the performance of a software product may in some cases be the determining factor in deciding which product to use. These are all compelling reasons to give the performance aspect a high priority in the design of a software system. The reality is, however, that performance has usually been relegated to the late life cycle stages in traditional software engineering. The popular belief has been that possible performance problems can be solved through code optimization and tuning. However, such an approach has several flaws: Not only can performance problems incur considerable costs to fix at such a late stage, but fixing performance problems through code optimations and tuning may also very well impair other quality traits of the system, such as the maintainabilty and flexibility of the system, since the code is made more complex and thus harder to understand. Finally, some performance problems are impossible to fix through code optimization and tuning - they reflect faulty design choices that can only be corrected through extensive re-design of the system.

A discipline called “Software Performance Engineering (SPE)” emerged in the eighties with the aim of trying to adress performance issues at the design time of systems, and by including explicit performance considerations already in the analysis phase of a development process. However, SPE has never really caught on in the mainstream software engineering practices. The reasons for this is that SPE relies on specialized formalisms for modelling the performance which are generally not being taught in the computer science courses at the majority of universities. This means that the amount of people that have the necessary knowledge to conduct SPE is limited, and SPE thus becomes inherently people-dependant.

In our master thesis we have proposed and investigated the validity of an approach that tries to incorporate early performance evaluation into software engineering without having to rely on specialized staff. The approach towards achieving this objective was to try and automate the process of obtaining input estimates which would otherwise have to be supplied by a performance specialist. The means for doing so is a test application that we have created as part of this project that captures performance characteristics of a target device. The test application can be used without any special performance knowledge to obtain input estimates. The input estimates returned from the test application are used to annotate activity diagrams with performance information according to the UML^{TM} Profile for Schedulability, Performance, and Time(UML-SPT). The annotated activity diagrams are then used to give a performance prediction of the system to be built.

Our work has concentrated on estimating performance for J2ME applications. More specifically we have targeted our effort on estimating performance for the two areas on the J2ME platform that typically constitute the greatest performance bottlenecks: persistent storage and networking.

To validate our approach we have applied it to a test case. We conducted a number of tests of our test application and test case to investigate whether the input estimates from the test application could be used as a basis for predicting the actual performance of the test case. The results from these tests showed that in the majority of cases this was fulfilled.

Our thesis is divided into three major parts:

In the first part of this thesis we investigated the existing work in the field of Software Performance Engineering and analysed the shortcomings hindering the adoption of SPE into the mainstream Software Engineering practices. Next, we described the performance relevant aspects of the UML-SPT profile in order to illuminate the concepts and extensions to the UML metamodel that allows for performance to be included in UML modeling.

We commenced the second part of the thesis with a description of our approach. We went on to illustrate how our approach is used as part of the design effort and the part concluded with an evaluation of the extent to which our approach is usable from a Software Engineering standpoint.

Finally, in the third part of this thesis we concluded on our work and discussed possible future work.

Bibliography

[1] Device specifications for nokia n70. `http://www.forum.nokia.com/devices/N70`.

[2] Device specifications for nokia n90. `http://www.forum.nokia.com/devices/N90`.

[3] Device specifications for nokia n91. `http://www.forum.nokia.com/devices/N91`.

[4] K. Auer and K. Beck. Lazy optimization: Patterns for efficient smalltalk programming. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[5] Doug Bell. Make java fast: Optimize! Internet: `http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html`.

[6] Christina Catley, Dorina C. Petriu, and Monique Frize. Software performance engineering of a web service-based clinical decision support infrastructure. In *Proceedings of the fourth international workshop on Software and performance*, pages 130–138. SIGMETRICS: ACM Special Interest Group on Measurement and Evaluation and SIGSOFT: ACM Special Interest Group on Software Engineering, ACM Press, New York, NY, USA, 2004.

[7] Simul8 Corporation. Simul8. Internet: `http://www.simul8.com/`.

[8] Vittorio Cortellessa and Raffaela Mirandola. Deriving a queueing network based performance model from uml diagrams. `http://portal.acm.org/citation.cfm?id=350406`, 2000. SIGMETRICS: ACM Special Interest Group on Measurement and Evaluation and SIGSOFT: ACM Special Interest Group on Software Engineering, ACM Press, New York, NY, USA.

[9] Vittorio Cortellessa and Raffaela Mirandola. Prima-uml: a performance validation incremental methodology on early uml diagrams. *Science of Computer Programming*, 44(1):101–129, 2002.

[10] Stig Dahl. Problembarnet amanda. Internet: `http://cph.ing.dk/arkiv/2400/amanda1.html`, 2000.

[11] Gentleware. Poseidon for uml. Internet: `http://www.gentleware.com/`.

[12] Eric Giguere. Record management system basics. Internet: `http://developers.sun.com/techtopics/mobility/midp/ttips/rmsbasics/`, 2001.

[13] The OMG Group. Uml profile for schedulability, performance, and time specification. January 2005.

[14] Gordon P. Gu and Dorina C. Petriu. Xslt transformation from uml models to lqn performance models. http://portal.acm.org/citation.cfm?doid=584369.584402, 2002. SIGMETRICS: ACM Special Interest Group on Measurement and Evaluation and SIGSOFT: ACM Special Interest Group on Software Engineering, ACM Press, New York, NY, USA.

[15] Gordon Ping Gu and Dorina C. Petriu. Early evaluation of software performance based on the uml performance profile. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, 2003.

[16] Stefan Haustein. Internet: `http://www.ksoap.org`.

[17] Stefan Haustein. Internet: `http://www.kobjects.org`.

[18] Jane Hillston. Pepa (performance evaluation process algebra). Internet: `http://www.dcs.ed.ac.uk/pepa/`.

[19] IBM. Rational rose xde developer. `http://www-306.ibm.com/software/awdtools/developer/rosexde/`.

[20] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[21] Robert F. Dugan Jr. Performance lies my professor told me: The case for teaching software performance engineering to undergraduates. In *Proceedings of the fourth international workshop on Software and performance*. Department of Computer Science, Stonehill College, ACM Press, New York, NY, USA, 2004.

[22] David P. Kelly and Robert S. Oshana. Software performance engineering a digital signal processing application. `http://portal.acm.org/citation.cfm?id=287328&coll=Portal&dl=GUIDE&CFID=44134284&CFTOKEN=22262372`, 1998. SIGMETRICS: ACM Special

Interest Group on Measurement and Evaluation and SIGSOFT: ACM Special Interest Group on Software Engineering, ACM Press, New York, NY, USA.

[23] M. Litoiu, Hamid Khafagy, Bin Qin, Anita Rass Wan, and J. Rolia. A performance engineering tool and method for distributing applications. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. IBM Centre for Advanced Studies Conference, IBM Press, 1997.

[24] Juan Pablo López-Grao, José Merseguer, and Javier Campos. From uml activity diagrams to stochastic petri nets: Application to software performance engineering. `http://portal.acm.org/citation.cfm?id=974048&coll=Portal&dl=GUIDE&CFID=44134284&CFTOKEN=22262372`, 2004. SIGMETRICS: ACM Special Interest Group on Measurement and Evaluation and SIGSOFT: ACM Special Interest Group on Software Engineering, ACM Press, New York, NY, USA.

[25] No magic Inc. Magic draw uml. Internet: `http://www.magicdraw.com/`.

[26] Daniel A. Menascé and Hassan Gomaa. On a language based method for software performance engineering of client/server systems. `http://portal.acm.org/citation.cfm?id=287331&coll=Portal&dl=GUIDE&CFID=44134284&CFTOKEN=22262372`, 1998. SIGMETRICS: ACM Special Interest Group on Measurement and Evaluation and SIGSOFT: ACM Special Interest Group on Software Engineering, ACM Press, New York, NY, USA.

[27] Daniel A. Menascé. Software, performance, or engineering? In *Proceedings of the third international workshop on Software and performance*. Department of Computer Science, George Mason University, ACM Press, New York, NY, USA, 2002.

[28] Ole Mertz, Kim Algreen, René Hansen, and Dennis Micheelsen. *.NET vs. Java - mobilklienter og web services*. The Faculty of Enginerring and Science - Department of Computer Science, Fredrik Bajers vej 7, 9210 Aalborg SOE, 2005.

[29] Visual Object Modelers. Visual uml. Internet: `http://www.visualuml.com/`.

[30] Object Management Group, Inc, `http://www.omg.org/technology/documents/formal/xmi.htm`. *XML Metadata Interchange (XMI)*.

[31] Object Management Group, Inc.(OMG), `http://www.uml.org/`. *UML 2.0 Superstructure Specification*, version 2.0 edition, October 2004.

[32] Stathis Papaefstathiou. State of the art of performance capabilites of commercial development environments. volume ACM WOSP 2002, `http://research.microsoft.com/~efp/WOSP%202002%20Tutorial.ppt`, 2002. Microsoft, Redmond, USA.

[33] Visual Paradigm. Visual paradigm for uml (vp-uml). Internet: `http://www.visual-paradigm.com/product/vpuml/`.

[34] University of Illinois at Urbana-Champaign PERFORM Performability Engineering Research Group. Möbius. Internet: `http://www.mobius.uiuc.edu/index.html`.

[35] Dorina C. Petriu and Hui Shen. Applying the uml performance profile: Graph grammar-based derivation of lqn models from uml specifications. Internet: `http://www.sce.carleton.ca/faculty/petriu/papers/TOOLS2002.pdf`, 2002.

[36] Roger S. Pressman. *Software Engineering. A Practitioner's Approach*. Alfred Waller. McGraw-Hill International (UK) Limited, 2000.

[37] Stanislaw Raczynski. Qms: Queueing model simulation. Internet: `http://www.raczynski.com/pn/qms.htm`.

[38] Andreas Schmietendorf, Evgeni Dimitrov, and Reiner R. Dumke. Process models for the software development and performance engineering tasks. In *Proceedings of the third international workshop on Software and performance*. ACM Press, New York, NY, USA, 2002.

[39] Peter Sestoft. Java performance: Reducing time and space consumption.

[40] James Skene. Unobtrusive performance analysis Ű where is the qos in tapas? Internet: `http://www.cs.ucl.ac.uk/staff/j.skene/tapas/TAPAS-pres-2.ppt#1`.

[41] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. ADDISON-WESLEY, 2002.

[42] Dr. Connie U. Smith. Spe-ed^{TM}. Internet: `http://www.perfeng.com/sped.htm`.

[43] Mesquite software. Csim. Internet: `http://www.mesquite.com/`.

[44] Ian Sommerville. *Software Engineering*. Addison-Wesley and Pearson Education Limited, 6 edition, 2001.

[45] Unknown. Queueing network model. Internet: `www.cse.msu.edu/~cse807/notes/slides/Queueconcepts-aga.ppt`.

[46] W3C, `http://www.w3.org/TR/xslt`. *XSL Transformations (XSLT) Version 1.0*, 1999.

[47] Pete Wildman. *Section 6 - Measures of Position*. STAT 2005, `http://wind.cc.whecn.edu/~pwildman/statnew/section_6_-_measures_of_position.htm`.

[48] Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. The Java Series. Sun Microsystems, Inc., `http://java.sun.com/docs/books/performance/`, 2000.

[49] Murray Woodside and Dorina Petriu. Puma - performance from unified model analysis. `http://www.sce.carleton.ca/rads/puma/`.