

Four-View Architecture for Software Performance Analysis using UML

Zhongfu Xu and Axel Lehmann

Institut für Technische Informatik
Universität der Bundeswehr München
85577 Neubiberg, Germany
email: xu, lehmann@informatik.unibw-muenchen.de

Abstract: As the unification of a few prominent software engineering methods, UML is gaining widespread acceptance and it is important for performance engineers to study the degree of adaptation of this modeling language for software performance analysis. In this work, we propose a use case-driven, UML-based software performance analysis approach which can be integrated into software development process. Constructed under a four-view architecture, the software models are capable of capturing the structural properties and quantitative behavior of software systems under development. The four-view software models are mapped to Queuing Network Models(QNMs) for software performance calculation. A case study demonstrates our approach with a simple ATM example.

1 Introduction

1.1 Unified Modeling Language(UML)

UML is a standard modeling language for writing software blueprints and can be used to visualize, specify, construct, and document the artifacts of a software-intensive system [BRJ99]. The main building blocks of UML are nine types of diagrams modeling the static and dynamic aspects of software systems. Some standard extensibility mechanisms are included in UML to enable the extension of UML in a controlled way. As the unification of a few prominent software engineering methods, UML is very expressive, addressing all the views needed to develop and then deploy software systems. It is appropriate for modeling software systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded software systems.

Although UML is intentionally quiet on software development process, one process that UML must enable is one that is use case-driven, architecture-centric, both incremental and iterative. This process is named as Unified Software Development Process [JBR99].

1.2 Unified Software Development Process (USDP)

USDP was developed hand in hand with UML and intended to be commonly followed by the whole software design society. Figure 1 shows the overview of USDP.

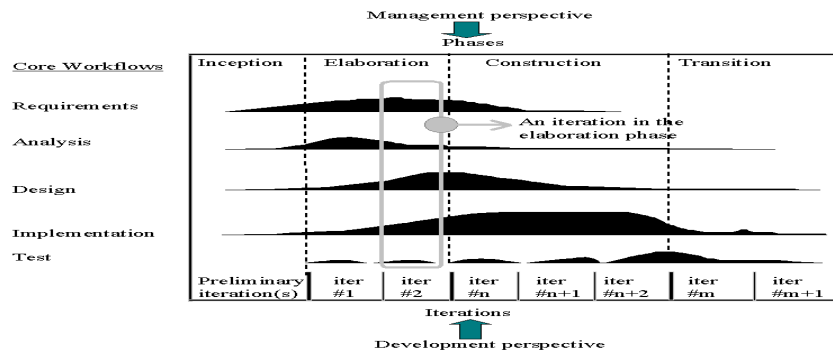


Figure 1: Overview of USDP [Ra01]

From a management perspective, the software development in USDP is decomposed over time into four sequential phases: inception, elaboration, construction, and transition. At each phase-end an assessment is performed to determine whether the objectives of the phase have been met. A satisfactory assessment allows the project to move to the next phase. Each pass through the four phases constitutes a development cycle producing a generation of the software.

From a development perspective, each phase in USDP is further broken down into iterations and the software development is a succession of iterations, through which the software develops incrementally. An iteration is in some sense one complete pass through all the workflows: Requirements, Analysis & Design, Implementation, Test, at least. It is like a small waterfall project in itself. Each iteration concludes with the release of an executable product.

1.3 Extend USDP workflows with performance analysis activities

In USDP, functional requirements (including feature sets, capabilities, and security) specify actions that a software system must be able to perform. Non-functional requirements (such as usability, reliability, performance, supportability) describe only attributes of the software system and system environment, and impose conditions on functional requirements. The functional requirements drive the USDP. As a non-functional requirement, software performance (throughput, response time, resource usage, and so on) does not gain the main focus. We think, USDP can be extended with performance analysis to ensure that the product of USDP has a better performance basis.

In order to be seamlessly integrated into USDP, the performance analysis process should have the same features as USDP. That is, it should also be use case-driven, architecture-centric, both iterative and incremental. Furthermore, the same tool, i.e. UML, should be used to reduce the cognitive gap between the two domains. The main advantage of the integration is the possibility to create both functional and performance models in parallel with the same design tools, and thereby make performance evaluation and control more effective and efficient. As a result, software performance is more likely to be improved along with the software development process and less likely to be viewed as "nice, but not part of the design".

In this work, we propose some performance analysis activities which can be added to the core workflows performed within the phases of each USDP cycle: **Identify and specify dominant use cases** and **define performance goals** to Requirements; **Identify the key computer resources** to Analysis; **Show the execution of each dominant use case** to Design; **Show the physical deployment of key computer resource instances** and **Map UML-based software models (UML diagrams) to QNMs and calculate performance measures** to Implementation; **Validate the performance models** to Test.

The rest of this paper is structured as follows: Section 2 introduces some related work on software performance analysis using UML. Section 3 details the UML-based performance-analysis-oriented activities we have proposed to extend USDP. Section 4 briefly introduces a case study to demonstrate the feasibility of our approach. In Section 5, some conclusions are made and our future work is outlined.

2. Some related work on performance analysis using UML

Since years, some work has been performed to use UML for performance analysis [Kä01] [HS98] [Ho00] [AS99] [AS00] [PK99] [Po99] [KP99] [KP98] [CM00]. The main problem with the related work is that in none of them, the integration of the performance analysis into functionality-oriented software development process is clearly demonstrated. They are mainly focused on the mapping of (parts of) UML-based design models to well-known formalisms in the field of performance modeling (i.e. Queuing Network[Kä01][PK99], Queuing Theory[HS98][Ho00], simulation model[KP98], Process Algebra[Po99], Petri Net[KP99], Execution Graph[CM00]). This way, the software performance analysis is essentially separated from software design process, and consequently, is post-tuning alike. Moreover, because of the lack of some performance-related information in design models, especially when the UML diagram types are only partially supported (e.g. in [Kä01], [AS99], [AS00], [PK99], [Po99], [KP99], [KP98], [CM00]), the mapped formalisms usually must be extended with performance-related information necessary for the calculation of performance measures and might lack effectiveness on complex software systems.

Our approach is aimed at the integration of software performance analysis with software design in USDP. The performance analysis begins early in and continue through the

whole software development process to ensure that the evolving software has a satisfactory performance. Under a four-view architecture, UML diagrams are used to build software models which are not only functionality- but also performance-analysis-oriented. The software models obtained are intended to contain all the performance-related information. Subsequently, the models are mapped to QNMs for calculation of performance measures. Due to the efforts made to include all the performance-related information in the software models, the computer-aided automatic generation of QNMs and QNM-based calculation of performance measures are achievable.

3. Software performance analysis activities using UML

In this Section, the performance-oriented activities briefly introduced in Section 1.3 are discussed in detail, producing UML-based models which are both functionality- and performance-oriented under an architecture of four views: **Task View** uses the UML use case and activity diagrams to describe the main tasks the software system must fulfil; **Resource View** describes the key computer resources participating in the software execution with UML class diagrams; **Behavioral View** describes the dynamic behavior of the computer resource instances by using UML sequence diagrams; **Deployment View** specifies the physical deployment of key computer resource instances using combined UML deployment and object diagrams. The standard extensibility mechanisms provided by UML, i.e. Stereotypes, Tagged Values, and Constraints, are used to represent the performance-related information or to make the resulted UML-based models more understandable. A mapping is made from these four views to a Queuing Network Model (QNM) on which the calculation of performance measures of interest is performed, as shown in Figure 2.

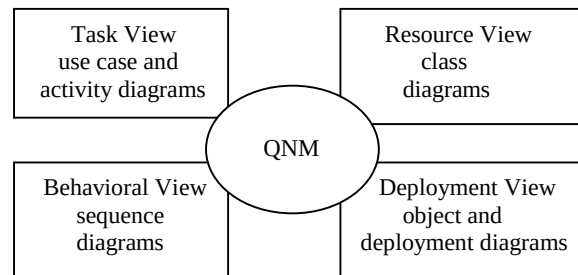


Figure 2: Four-view architecture for performance analysis

3.1 Identify and specify the dominant use cases→ Task View

The proposed performance analysis process is use case-driven. That is, use cases are the primary input for performance analysis. Performance analysis could be made for all the use cases of the software system. However, in order that the performance analysis process is more manageable, not all the use cases in the software system under development are

being considered. Only the dominant ones that have decisive influence on the performance of the software system are taken into consideration.

The dominant use cases either might be very frequently requested by the users, and thus drive the performance of the entire system, or might have great demand for the computer system resources during the execution, or both. Having been identified, they are represented with UML **use case diagrams**. The rate at which these dominant use cases are requested and their request patterns are also specified, for example, by using UML Tagged Values and Constraints. The participation of the user representatives is necessary for this work. UML **activity diagrams** are drawn for the dominant use cases to help understanding which computer resources take part in the execution of the use cases.

For these dominant use cases, performance goals are defined. Performance goals are the criteria for evaluating the software system plans and the performance of the software product. They must be specific and measurable by defining how the goal achievement will be measured, by defining all the influential factors. Quantifiable performance goals provide the basis for selecting appropriate design alternatives. They should also be realistic and cost-effective. The persons involved in the project should collaborate to clarify the performance goals. They could define performance measures (for example, response time, throughput, and resource budgets in terms of average, maximum, percentile, and so on), the measurement environment (peak load, daily average, number of concurrent users, or other factors), and even the events to be measured.

3.2 Identify the key computer resources→ Resource View

The Resource View uses UML **class diagrams** to specify the computer resources involved in the execution of identified use cases.

Active computer resources, such as processors, I/O devices, are represented as classes and interfaces in UML class diagrams. They are classified into two categories and rendered as stereotyped classes: <<queue server>> and <<delayer>>. The queue server consists of two elements: queue and server. The jobs get through the queue to the server for processing and wait in the queue when the server is busy providing service for another job. The delayer can process multiple jobs at one time.

The passive computer resources, such as memory, files, database, usually are not explicitly modeled. Their influence on the use case execution is considered during modeling of the active resources that directly manipulate them¹. For example, the reading and writing operations on memory are specified as operations performed by CPUs. In some cases, the passive resources are exclusively used by other computer resources for a long time or frequently. This may lead to a situation with only one process at a time point able to execute, and then should be modeled explicitly. To model the exclusive usage of passive resources, we introduce two logical resource classes *Allocator*---queue server and

¹ The passive resources should be carefully considered when the performance bottleneck analysis shows the corresponding active resources are bottlenecks.

Releaser---server without queue. The passive resources are then modeled as server without queue. A transaction arrives at an Allocator and requests a token representing the passive resource. If a token is not available, the transaction waits in the queue of the Allocator. The transaction leaves the queue when its request is granted. It keeps the token until it reaches a Releaser(or until it leaves the system). The Releaser takes the token from the transaction and gives the token to the designated Allocator.

Furthermore, not all the involved active computer resources are identified, but only the key resources that are either indispensable for software execution or provide service in a relative low speed. By modeling a minimum number of active computer resources adequately characterizing software performance and hiding the passive resources, the formulation and evaluation of the performance model will be simplified.

Submodels can also be used to aggregate some resources, such as network resources in distributed-object systems, within one flow-equivalent server. When the performance analysis indicates situations that warrant further study, the submodels can be expanded to a more detailed level.

The performance-related characteristics of the identified computer resources are specified in form of attributes of corresponding classes. These attributes describe the capabilities of the computer resources during software execution and will be used by performance model solution tools to calculate performance measures. Some attributes are commonly needed for queue servers, such as queuing discipline, service capability(for example, average ratio between a high level instruction and a machine instruction and MIPS for CPUs), distribution of service time, maximum queue length, and so on. The important performance-related attribute of delayers is the delay time that jobs need for their services.

When the values of some attributes are known to be unchanged for all instances of a class, these attributes are defined to be static with fixed values. These values are obtained, for example, by (re)doing the environment analysis, measurement, benchmark tests, and can be specified using approximate values, bounded values(best- and worst- case), or intervals². Some attributes might be dependent on the dynamic behavior of the computer resources during software execution. They will be specified during modeling the use case execution(with UML sequence diagrams).

Besides the attributes, the operations of classes(methods) and the contents of the interfaces (abstract methods) are also defined. They specify the execution behavior of the computer resources, i.e. what they do during software execution. We think it is important that the methods are appropriately fine-grained, more clearly, they should be *atomic*, i.e. they have a definite execution time and can fulfil their tasks alone, without the help of other methods. Keeping the methods to be atomic can reduce the ambiguity when specifying the software execution behavior (with UML sequence diagrams). For example,

² Some interval-based methods[LH98] [LK00] can be used to solve the resulted performance models

the message sender and receiver are easier to specify. Another advantage is that the investigation of processing overheads of methods is more achievable.

In addition, for software systems to be modeled with open QNMs, the source of job arrival (infinite or finite) is specified as a class with descriptions about its job arrival process (and average job arrival rate), or inter-arrival time distribution. For closed QNMs which have a fixed population circulating among the stations, the job source is defined as a class with the number of jobs and the average think time.

3.3 Show the execution of each dominant use case→ Behavioral View

In Behavioral View, the UML **sequence diagram** is used to show the interactions among objects, emphasizing the time-ordering of the actions performed by objects. The objects are instances of the classes in Resource View, representing key computer resources participating in the execution of the use cases. They will be mapped to the stations of a QNM. In order to eliminate ambiguity, the instances of the same resource class are distinguished by naming them differently or using annotations. During specifying the time-ordering of interactions among objects, the sequence diagram shows the flows of control within the execution of a use case.

After modeling the software execution behavior using UML sequence diagrams, the service demands of actions performed by computer resources are investigated when they have not been specified in class diagrams in Resource View. Usually, the service demands indicate the actual time during which the corresponding computer resources perform some work and are specified as service time (for example, for an I/O operation). Some are specified in other forms (such as the number of high level instructions executed by CPUs, and then, the service capability parameters of CPUs specified in Resource View are used to obtain the service time for each operation performed by CPUs).

Among others, some techniques can be applied to investigate the service demands of the actions performed by each computer resource instance, such as measurement, using software libraries containing standard and well-tested software routines, performance benchmarking, prototypical implementation, using UML state-chart diagrams to model the event-ordered behavior of an individual object, rules of thumb, etc. In most cases, approximations are sufficient to identify critical areas for which precision is vital[Sm90]. The approximations of service demands can be represented, for example, as ballpark numbers, bounded values(best- and worst-case values, and intervals).

A usual case is that the service times of some operations are dependent on some factors. For example, the network transmission time depends on the size of data to be transmitted. The influencing factors may differ each time the software executes and cause variability of the service time of the same operation. The variability makes the specification of service times more difficult. One solution to the variability problem is to specify the service time with a distribution or with an interval. A cheaper, approximate solution is to specify the service time with a fixed value, such as average value, best- and worst-case value. The latter solution might be reasonable when it is known that the service time to be

specified (for example, the service times needed by a very fast processor) is relatively low compared with the service times of the operations of other resources. Both solutions require the knowledge about the influencing factors.

The service demands of actions performed by each object are added to the Behavioral View by rendering them as annotations when they have not been specified in Resource View. This is achieved by using Constraints and Tagged Values, the standard extension mechanisms provided in UML. The UML **object diagrams** can also be used here to show the relationships of the objects, their attribute values and processing overheads of their operations.

3.4 Show the physical deployment of key computer resource instances → Deployment View

The UML deployment diagrams are usually used to show the configuration of run-time processing nodes and the components that live on them. They are capable of capturing the topology of hardware platform. Nevertheless, both nodes and components in the deployment diagram are usually *larger* than the stations in a QNM. As a result, a mapping from a deployment diagram without extension to a QNM is usually impossible.

In order to directly map the deployment diagram to a QNM, the deployment diagram is extended by adding the processes and objects running in the components. That is, the **deployment diagram** and **object diagram** are combined to show the physical deployment of key computer resource instances. The resource instances (objects) are distributed in the run-time nodes and will be mapped to the stations in the QNM together with the interconnections among them.

3.5 Map UML-based software models to QNMs

The performance-related information is distributed in various forms in the UML-based software models produced during software development. This causes difficulty for performance calculation. As a solution to this problem, a transient *information container* should be found to centrally hold these performance-related information. This container is used as the information source for performance calculation. As a classic approach for modeling and evaluation of software performance, QNMs are capable of this work. QNM-based performance models can easily identify performance problems and point to remedies[Sm90]. Moreover, there is a variety of well-tested methods and tools supporting the solution and evaluation of QNM-based performance models.

In order to build a QNM, information is generally needed about:

- Stations

The QNM stations usually represent physical computer resources participating in the software execution. Their properties, such as delay time, service time distribution, queuing discipline, queue capacity, are necessary for performance calculation. In

order to model some special software system behavior (such as exclusive usage of passive resources, conditional execution, iterative execution, synchronization, forking/joining and splitting of execution, simultaneous resource possession), some of the QNM stations represent logical computer resources, for example, Allocators and Releasers to model exclusive usage of passive resources.

- **Arcs**
Arcs interconnect the QNM stations and indicate the flows of control within the QNM.
- **Workloads**
The workloads represents the amounts of work placed upon the QNM stations by the software and its users. They should be specified explicitly, for example, in terms of actions performed by each QNM station and their service demands. The workload for the entire QNM should also be determined by defining the job arrival process or job inter-arrival time distribution(for open QNMs), or the number of jobs in the system and the average think time of the user(for closed QNMs).

The information needed for building a QNM is contained in the four-view, UML-based software models:

- The objects in Behavioral View (i.e. sequence diagrams) and Deployment View (i.e. combined deployment and object diagrams) actually participate in software execution. They are mapped to the stations of the QNM. Their properties are specified in Resource View (i.e. class diagrams) and Behavioral View.
- The relationships among objects implied in Behavioral View and explicitly shown in Deployment View indicate the interconnections between QNM stations. A navigable (bi-directional) association between two objects in Deployment View is mapped to two directed arcs between the two corresponding QNM stations. A single-directional association in Deployment View is mapped to one directed arc in the QNM. The QNM arcs originate always from a server or a delayer and terminate in a queue of a queue server, a server without queue or a delayer.
- The QNM workloads are described in Task View(i.e. use case and activity diagrams) and Resource View, and detailed in Behavioral View. These three views specify what the software system does and how it does its work(i.e. time-ordering and processing overheads of actions performed by the computer resources).

3.6 Calculate performance measures

The QNM-based performance models can be solved by using simulation, analytic approaches, or numerical methods [Li98] [HS98]. The performance measures calculated are used to check whether the predefined performance goals for the dominant use cases are met. When goals are met, the software development evolves. Otherwise, performance bottleneck analysis is carried out and design alternatives for performance improvements are recognized by, for example, changing the user request pattern, redesigning the use case execution path, moving some processing work from the bottleneck to another computer resource, exploiting parallelism between activities, caching data to reduce

storage access, using more resources, or adopting faster devices. Then, performance analysis processes are redone to evaluate the design alternatives. In some cases, performance goals are modified to be more realistic. The UML-based software models are validated by comparing the calculated performance measures of these dominant use cases with the test results produced by software testing.

4. A case study [XLL01]

We assume the customers of a bank use an Automated Teller Machine(ATM) to check account balance, withdraw from and deposit money to accounts. In this work, we choose the **Get-Balance** use case to demonstrate our proposal. Here, we consider only the complete execution of the use case without exceptions. We will investigate the average response time of this use case, the average throughput and utilization of each involved computer resource. As the main performance criterion, the average response time should be less than 150 seconds with a percentile of 90%. For this use case, four-view models are constructed in [XLL01] using UML.

The Resource View in Figure 3 shows the identified computer resources and their relationships, together with their attributes and atomic actions. We assume that only one ATM is in use. Since a user will not be interrupted when he interacts with the ATM, the *Console* is used exclusively. Two logical classes *AllocatorForConsole* and *ReleaserForConsole* are introduced to model the exclusive usage of Console.

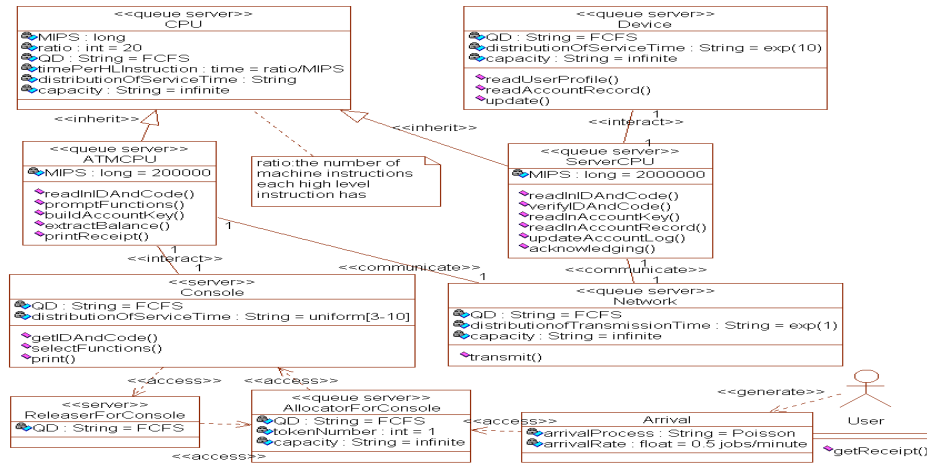


Figure 3: Resource View for Get-Balance use case

The Behavioral View in Figure 4 shows a typical sequence of actions performed within the Get-Balance use case. Using UML constraints, the operations performed by *ATMCPU* and *serverCPU* are annotated with the assumed numbers of executed high level instructions and the service times in braces.

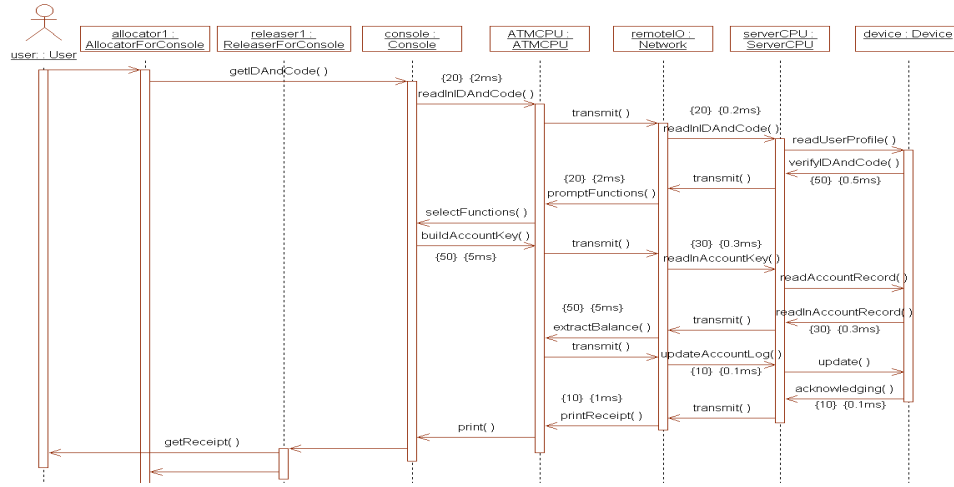


Figure 4: Behavioral View for Get-Balance use case

The four-view software models for Get-Balance are mapped to a QNM shown in Figure 5. This use case is modeled as an open QNM: Users arrive at *allocator1*; When the *console* is not in use, *allocator1* locks the *console*, and the processing begins; Upon the completion of processing (i.e. the required bank balance is printed), *releaser1* frees the *console* for the use by another user.

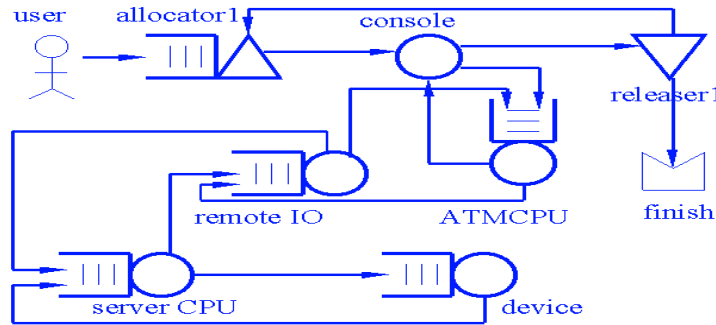


Figure 5: QNM for Get-Balance use case

Using the commercial simulation tool *SIMUL8* by Visual Thinking International Ltd., a simulation experiment is carried out on the QNM in Figure 5 with a Results Collection Period of 600000 seconds. Totally, 5129 jobs are generated. The average response time of each job is 89.77 seconds with a standard deviation of 49.65 seconds. The minimum response time is 46.53 seconds, and the maximum is 598.84 seconds. 90% of the jobs have response times within 150 seconds. Thus, it is reasonable to think that the predefined performance goal is achieved. The maximum length of user queue before *console* (i.e. in the queue for *allocator1*) is 10, the average queuing time of users is 30.43 seconds, the

maximum queuing time is 542.88 seconds, and 66% of the queuing times are within 30 seconds. The performance measures for computer resources are summarized in Table 1.

Table 1

	console	ATMCPU	RemoteIO	ServerCPU	Device
utilization	0.167	0.0001	0.051	0.0008	0.0026
throughput	0.026	0.043	0.051	0.051	0.026

5. Conclusions and future work

We have proposed a use case-driven, UML-based software performance analysis approach which is intended to be integrated into the Unified Software Development Process. In our approach, UML-based software models which are both functionality- and performance analysis-oriented are constructed under a four-view architecture. For performance calculation and analysis they are mapped to QNMs. A case study demonstrates our approach using a simple ATM example, and suggests that four-view UML-based models are capable of capturing the structural properties and quantitative behavior of software under development.

The main limitation of our approach is that the proposed mapping from four-view software models to QNMs is not automated, and might be labor-intensive for the modeler, especially when the analyzed software system is of large scale. Our research will continue to automate the mapping from the four-view software models to QNMs for performance calculation. The automated mapping would be achievable with help of some UML tools, such as Rational Rose 2000 of Rational Software[Ra01]. UML-based software models produced by Rational Rose 2000 can be saved in XML Metadata Interchange(XMI) format using XMI support from Unisys Corporation. An algorithm would then be developed to extract all the performance-related information from the resulted XMI document to construct the QNMs for performance calculation.

References

- [AS99] Arief, L.B.; Speirs, N.A.: "Automatic Generation of Distributed System Simulations from UML", *Proc. 13th European Simulation Multiconference(ESM'99)*, Warsaw, Poland, pp.85-91, June, 1999.
- [AS00] Arief, L.B.; Speirs, N.A.: "A UML Tool for an Automatic Generation of Simulation Programs", in *Second Int. Workshop on Software and Performance(WOSP2000)*, Ottawa, Canada, September 18-20, 2000.
- [BRJ99] Booch, G.; Rumbaugh, J.; Jacobson, I.: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

- [CM00] Cortellessa, V.; Mirandola, R.: "Deriving a Queuing Network based Performance Model from UML Diagrams", in *Second Int. Workshop on Software and Performance(WOSP2000)*, Ottawa, Canada, September 18-20, 2000.
- [Ho98] Hoeben, F.: "Using UML Models for Performance Calculation", in *Second Int. Workshop on Software and Performance(WOSP2000)*, Ottawa, Canada, September, 18-20, 2000.
- [HS98] Hoeben, F.; Sterk, M.: "Performance in Distributed Object Systems", Software Engineering Research Center, Utrecht, The Netherlands, 1998.
- [JBR99] Jacobson, I.; Booch, G.; Rumbaugh, J.: *The Unified Software Development Process*, Addison-Wesley, 1999.
- [Kä01] Kähkipuro, P.: "UML-Based Performance Modeling Framework for Component-Based Distributed Systems", R. Dumke et al.(Eds.): *Performance Engineering, LNCS 2047*, pp.167-184, Springer-Verlag Berlin Heidelberg 2001.
- [KP98] Kabajunga, C.; Pooley, R.: "Simulation UML Sequence Diagrams", in *R.Pooley and N. Thomas editors, UK PEW'98-Proceedings of 14th UK Performance Engineering Workshop*, Edinburgh, July 1998.
- [KP99] King, P.; Pooley, R.: "Using UML to Derive Stochastic Petri Net Models", in *N. Davies and J. Bradley, editors, UK PEW'99-Proceedings of the 15th UK Performance Engineering Workshop*, Department of Computer Science, the University of Bristol, July 1999.
- [LH98] Lüthi, J.; Haring, G.: "Mean Value Analysis for Queuing Network Models with Intervals as Input Parameters", *Performance Evaluation*, Vol.32, No.3, pp185-215, April 1998.
- [Li98] Lindemann, C.: *Performance Modeling with Deterministic and Stochastic Petri Nets*, John Wiley&Sons, 1998.
- [LK00] Lüthi, J.; Kladó, C.M.: "Splitting Techniques for Interval Parameters in Performance Models", Technical Report No. 2000-07, Fakultät für Informatik, Universität der Bundeswehr München, 85577 Neubiberg, Germany, Dec. 2000.
- [PK99] Pooley, R.; King, P.: "The Unified Modeling Language and Performance Engineering", *IEE Proceedings-Software*, Vol. 146, No. 1, pp.2-10, February 1999.
- [Po99] Pooley, R.: "Using UML to Derive Stochastic Process Algebra Models", in *N. Davies and J. Bradley, editors, UK PEW'99-Proceedings of the 15th UK Performance Engineering Workshop*, Department of Computer Science, the University of Bristol, July 1999.
- [Ra01] Rational Software Corporation, <http://www.rational.com>, 2001.
- [Sm90] Smith, C. U.: *Performance Engineering of Software Systems*, Addison-Wesley, Reading, Massachusetts, 1990.
- [XLL01] Xu, Z.; Lehmann, A.; Lüthi, J.: "Extending Unified Software Development Process with Performance Analysis using UML", Technical Report, Fakultät für Informatik, Universität der Bundeswehr München, 85577 Neubiberg, Germany, 2001.