UNIVERSITÀ CA' FOSCARI DI VENEZIA
Dipartimento di Informatica
Technical Report Series in Computer Science

Rapporto di Ricerca CS-2004-2

Aprile 2004

M. Marzolla, S. Balsamo

UML-PSI: the UML Performance SImulator

Dipartimento di Informatica, Università Ca' Foscari di Venezia
Via Torino 155, 30172 Mestre–Venezia, Italy

UML-Ψ: the UML Performance SImulator

Moreno Marzolla Simonetta Balsamo

Dipartimento di Informatica
Università Ca' Foscari di Venezia
via Torino 155, 30172 Mestre (VE), Italy
e-mail: {marzolla|balsamo}@dsi.unive.it

Technical Report CS-2004-2, April 2004

Abstract

In this paper we describe UML-Ψ, a software performance evaluation tool based on process-oriented simulation. The tool can be used to evaluate performances of software systems which are described at a high level of abstraction. This allows the software modeler to estimate the system performances before actually building it, at the design phase. We consider software specifications as Unified Modeling Language (UML) diagrams annotated according to a subset of the stereotypes and tagged values defined in the UML Performance Profile. UML-Ψ transforms the software model, i.e., the set of UML annotated diagrams, into a performance process-oriented simulation model. Simulation parameters are derived by the software description and the additional information given in the specification. Simulation experiments are carried on by choosing an appropriate condition for the length of the simulation run. The tool collects observations and derives estimates of a set of performance measures. Performance results are interpreted at the software specification level, as they are inserted into the software model as tagged values associated to the relevant UML elements. Further analysis can be applied to the possibly modified software model in order to meet given performance requirements. We illustrate with an example how the UML-Ψ tool can be used.

Keywords Unified Modeling Language, Simulation Modeling, Software Performance Evaluation

1 Introduction

Quantitative analysis of software systems is being recognized as an important issue in the software development process. Performance analysis can help to address quantitative system analysis from the early stages of the software development life cycle, e.g, to compare design alternatives or to identify system bottlenecks. Early identification of performance problems is desirable as the cost of design change increases with the later phases in the software development cycle.

Performance evaluation should be highly integrated as possible in the software development process [4, 19, 20]. This means that the software engineer should be provided with integrated performance and specification environments where the performance evaluation tools should provide the following characteristics:

- *No special performance modeling expertise required*: the tool should be easy to use, requiring little or no specific performance modeling knowledge and skills from the user. The performance evaluation tool should be integrated within a CASE tool.
- *Automation*: the integrated software performance tool should be mostly automatic; the user interaction should be minimized, i.e. the user should not be required to perform actions or computations by hand.
- *Feedback*: the performance results should be reported at the Software Architecture (SA) level.

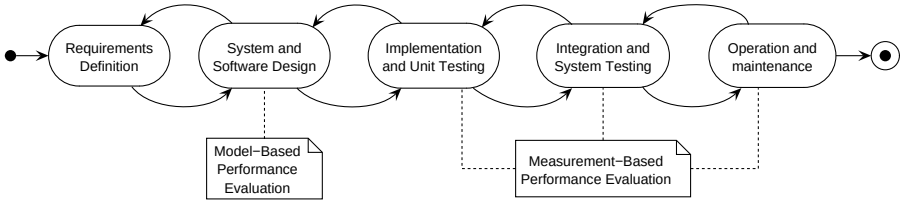


Figure 1: The Waterfall Software Development Model

We propose a model-based performance evaluation approach. This consists on deriving a performance model from a high-level software description at the software design phase of the software development cycle, as shown in Fig. 1. The advantage is that performance can be evaluated without building and measuring an actual system implementation; this would be required by measurement-based techniques.

In this framework we describe the simulation-based software performance tool UML-Ψ, which uses UML software specifications and process oriented simulation as the performance model. The tool is the application of a software performance modeling approach, as described in [5, 12], defined for software architecture performance analysis. The approach integrates UML software specification given by a set of annotated diagrams, with a discrete-event simulation model whose solution gives a set of average performance indices providing feedback at the software architectural level. The tool is a software package which can be used by the performance modeler to generate and evaluate the performance model, as shown in the boxed Use Cases of Fig. 2.

The integrated software and performance modeling approach identifies two main roles for the users interacting with the system: software modelers and performance modelers; note that these roles may both be played by the same individual.

The software modeler is responsible for:

- Designing the system, given a set of functional and non-functional specifications and requirements.
- Defining the performance requirements which should be met, as part of the non-functional specifications.
- Interpreting the figures of merit derived from the evaluation of the software performance model.
- Modifying the software model if the performances requirements cannot be satisfied.

The performance modeler is responsible for analyzing and evaluating the software model. In particular, the performance modeler can:

- Annotate the software model with quantitative, performance-oriented informations.
- Generate the performance model from the annotated software specification.

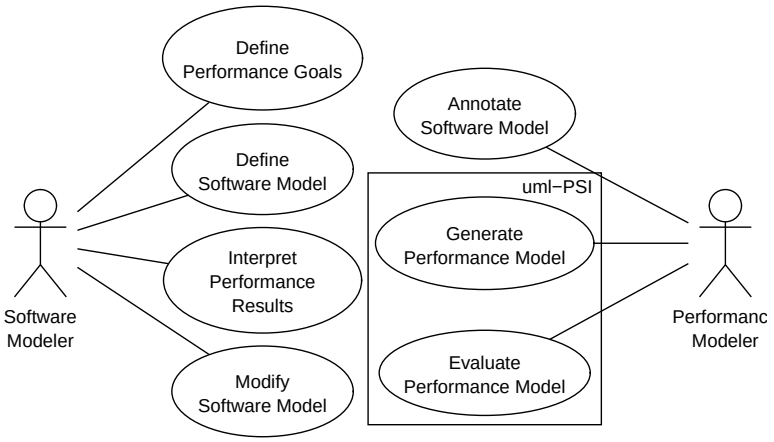


Figure 2: Usage scenarios of UML-Ψ

- Evaluate the performance model, obtaining a set of figures of merit for the software architecture.

The software model is drawn, annotated and modified using a UML CASE tool which must be capable of exporting the model in XML Metadata Interchange (XMI) format [14]. The tool considers the following UML diagrams to derive the performance model: Use Case, Activity and Deployment diagrams. UML-Ψ parses the annotated UML model, which must be exported in XMI format from the CASE tool, and builds a process-oriented simulation model of the software system. The simulation model implementation, e.g., the simulation program is executed, and computes a set of performance indices of the software system under study, such as resources utilization and throughput, and the mean execution time of actions and Use Cases. Simulation results are reported back into the original software model as UML tagged values associated to the relevant elements. This is allows us to give an user-friendly feedback. A schematic representation of how UML-Ψ can be used is given in Fig. 3.

Figure 3: Using UML-Ψ

Fig. 4 illustrates the high-level structure of the UML-Ψ model processing framework. The UML model is created using a CASE tool, and is exported in XMI format. XMI is an XML-based notation for describing UML models, useful for exchanging UML models among different applications. Due to incompatibilities among different implementations of the XMI format, UML-Ψ currently supports the XMI dialect used by the open-source ArgoUML [2] tool version 0.12, and its commercial counterpart Poseidon [17] version 1.4. Supporting other CASE tools is possible, by simply implementing an appropriate XMI parser module.

UML-Ψ parses Use Case, Activity and Deployment diagrams in order to build an internal representation of the UML model. A process-oriented simulation model is then derived from this internal representation. The simulation model has the same structure as the software specification. i.e., the performance model instantiates simulation processes to model action states, active or passive resources and workloads. A more detailed description of the simulation model will be given in the next section.

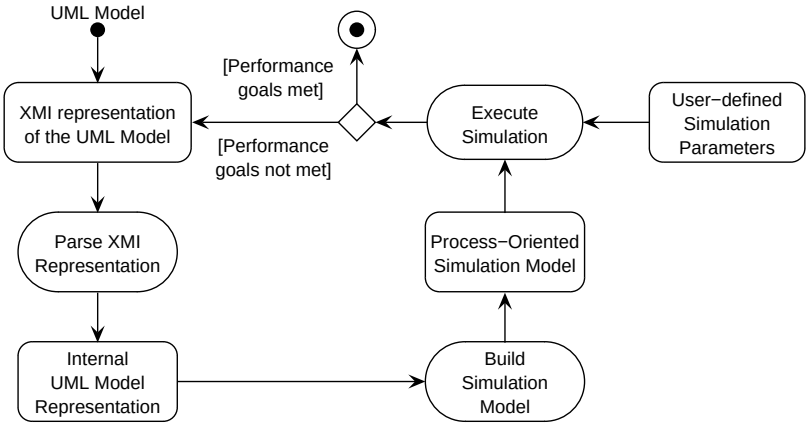


Figure 4: The UML-Ψ model processing framework

The UML-Ψ tool executes the simulation performance model by using both the user specified parameters, that are given as tagged values associated to UML elements, and the parameters included in a configuration file. Simulation parameters include, but are not limited to, the number of times an action is repeated, the service demand of actions, expressed as random variables with a given distribution, scheduling policies of active resources and others. See [12] for a complete description of the available tag names and values. We consider the specification of tag values by the Tag Value Language (TVL), a subset of the Perl language [23] proposed in [13]. This is motivated by the need to express such values in a complex way: for example, it may be required for specifying relationships between tagged values. Hence, it is necessary to be able to manage expressions, such as arithmetic or boolean ones. Since TVL is defined in [13] as a subset of the Perl language, there are two main advantages: it is likely to be familiar to many users, because Perl is a widely used language, and Perl is supported by a wide range of readily available open source tools and utilities. We took a step further, and decided to embed a full Perl interpreter inside UML-Ψ, as one is freely available [16].

Another advantage of using a programming language to describe tag values is that the software model may be parameterized. This allows the modeler to develop a single UML specification, where some performance parameters are left unspecified. The same model may be analyzed for different parameter values by specifying different configuration files. In UML-Ψ, a configuration file is a user-defined Perl program. UML-Ψ parses the program and uses the Perl interpreter environment (modified by every declaration contained in the configuration file) to parse tag values. If a tag value contains an expression such as:

```
PAdemand = [‘‘assm’’,’’dist’’,[‘‘exponential’’, $mean]]
```

the configuration file must specify a value for `$mean`, such as for example:

```
$mean=5.0;
```

The simulation performance model is eventually executed and the computed results are inserted into the XMI document as tagged values associated with the UML elements they refer to. Such results are available to the user which can open again the UML model by using the CASE tool. The performance modeling cycle illustrated in Fig. 4 may be iterated to meet given performance requirements.

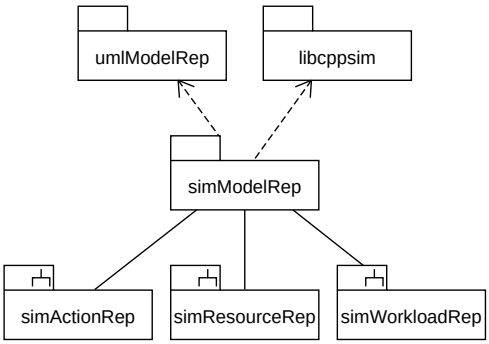


Figure 5: The UML-Ψ tool structure

Related works To the best of our knowledge, UML-Ψ is one of the first tools based on a subset of the annotations proposed in the UML Performance Profile [13]. Some tools for performance evaluation of software systems have been recently proposed, starting from the first approach the Software Performance Engineering first described by Smith [19]. A survey and classification of the methodologies and some tools can be found in [4].

A few approaches based on simulation performance models have been proposed. Arief and Speirs [3] developed a tool for deriving simulation models from UML class and sequence diagrams. The approach translates UML diagrams into a simulation model described as an eXtensible Markup Language (XML) document. Then the method implements the model as a simulation program, which is executed and provides performance results. The advantage of this approach is the ability to decouple the simulation model from its implementation, which makes it possible to implement the simulation model using different languages.

De Miguel et al. [7] introduce UML extensions for the representation and automatic evaluation of temporal requirements and resource usage, particularly targeted at real-time systems. These extensions are based on a set of formal constraints, tagged values and stereotypes. The simulator is based on the Objecteering CASE Tool [21] and OPNET Modeler [15].

Hennig et al. [8] describe a UML-based simulation framework for early performance assessment of software/hardware systems described as UML Sequence and Deployment diagrams. The simulation model is implemented using the discrete-event simulation package OMNeT++ [22].

The UML-Ψ simulator differs from the tools above because the underlying software performance modeling approach is different. Our approach is based on the UML Performance Profile [13], which defines a set of standard annotations to specify performance-oriented informations in a standard way. UML-Ψ is not tied to any particular CASE tool (only the module responsible for parsing the XMI representation is), which allows a greater degree of flexibility.

2 UML-Ψ internal structure

UML-Ψ is a tool written in C++ for performance and portability reasons. It has been tested on the Linux operating system, but should be easily ported to most Unix platforms with small or no changes. The UML package diagram of Fig. 5 illustrates the high-level structure of UML-Ψ.

The umlModelRep and simModelRep packages contain classes used for the representation of the UML model and the process-oriented simulation model, respectively. Note that the software perfor-

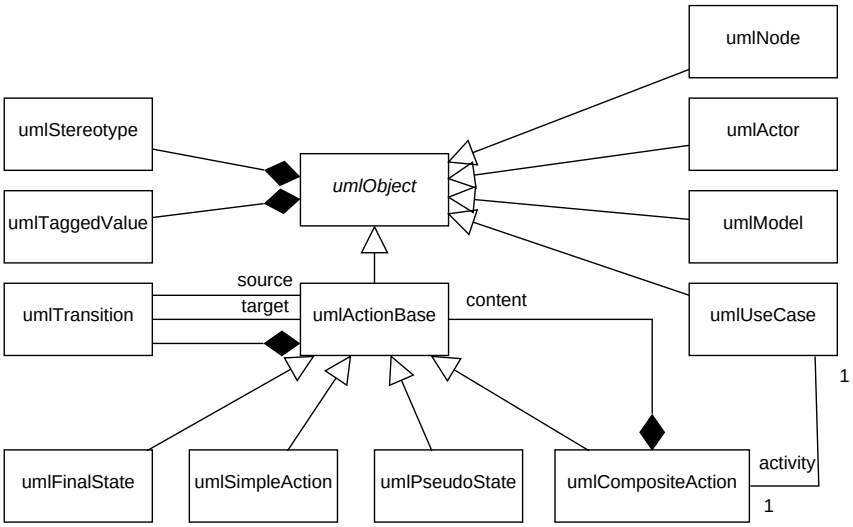


Figure 6: UML-Ψ class diagram, representation of the UML metamodel

mance evaluation approach associates the processes of the simulation model to the UML model objects. Therefore the simModelRep package depends on umlModelRep. Moreover, the simModelRep package depends on libcppsim, which provides basic functionalities for process-oriented simulation modeling and simulation output data analysis. The simulation model is based on three main types of compoents or entities, corresponding to the actions of activity diagrams, the resource of the software systems and the workloads. Then, the simModelRep package contains three sub-packages, which include simulation classes representing actions (simActionRep), resources (simResourceRep) and workloads (simWorkloadRep).

2.1 UML model representation

Fig. 6 shows the class diagram of the software specification UML model. UML-Ψ parses an XMI representation of the annotated UML model, extracting from the model only those diagrams which are used for building the simulation model. These include:

- Use Case diagrams, used to represent the workloads applied to the system;
- Deployment diagrams, used to model the available physical resources, both active and passive ones;
- Activity diagrams, used to model the computation of the software system.

The software model must be annotated according to a subset of the annotations described in the UML Performance Profile [13]. The annotations are based on the standard UML extension mechanisms of stereotypes and tagged values. We define stereotypes and associated tagged values for describing open or closed workloads, active or passive resources, and computational steps. A detailed description of the annotations can be found in [5, 12].

The root class used for the internal representation of the UML model is umlObject. This class contains the common attributes and basic operations of all UML model elements. Each UML object can be stereotyped with at most one stereotype, and may have an arbitrary number of tagged values associated with it. umlStereotype and umlTaggedValue represent stereotypes and tagged values objects, respectively. Class umlModel represents the entire UML model, which contains objects of the three classes

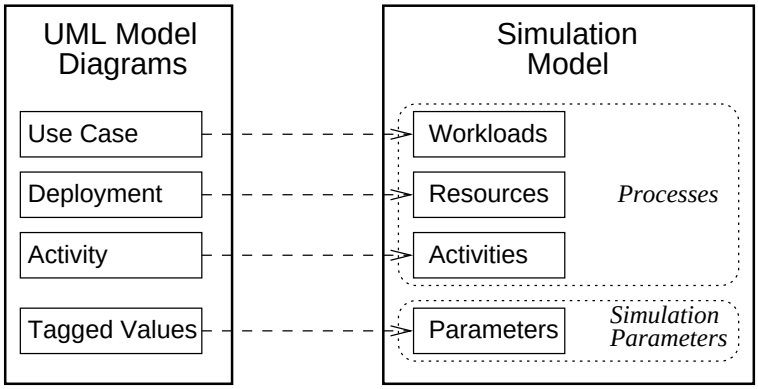


Figure 7: Mapping UML elements into simulation processes

umlActor, umlUseCase and umlNode. They represent Actors, Use Cases and Deployment diagram nodes, respectively. A Use Case is associated with a composite action which describes its behavior.

The class umlActionBase represents the common features of UML actions. This abstract class contains a reference to a list of outgoing transitions (umlTransition). A transition, in turn, is associated to a source and a target action. The four kinds of actions in UML models are represented by the class inheriting from umlActionBase, as shown in fig. 6. UML composite actions are represented by the umlCompositeAction class; a composite action is an action which can be furtherly decomposed in other sub-actions at a greater level of detail. A simple (atomic) UML action state is represented by the umlSimpleAction class. Finally, UML pseudo states (fork and join nodes) are modeled by the umlPseudoState class. The final state is represented by umlFinalState.

2.2 Simulation model representation

In this section we describe the structure of the simulation model. We define three kinds of simulation processes: processes representing workloads, resources and activities respectively. As shown in Fig. 7, UML actors are translated into workloads; nodes of the deployment diagrams correspond to processes modeling resources, and action states in activity diagrams are translated into processes representing the actions. UML annotations are used as parameters for the simulation model.

In Table 1 we list all the processes types used in the simulation model. The table shows the process names in the left column, and the list of other processes with which it can interact.

An brief description of each process is given as follows:

OpenWorkload This process performs an endless loop generating an infinite stream of processes of type OpenWorkloadUser. After each process is created, it pauses for a random amount of simulated time to simulate the interarrival time between requests.

OpenWorkloadUser Represents a request arriving to the software system. It triggers the activation of one of the use cases associated to the actor representing the workload, and then terminates. The use case is modeled as a CompositeAction process.

ClosedWorkload This process creates a fixed population of system requests, represented by processes of class ClosedWorkloadUser. All the requests are activated, and the ClosedWorkload terminates.

ClosedWorkloadUser Represents a request of service to the software system. The request belongs to a finite set of requests generated from the same closed workload. This process behaves like a

Process Name	Interacts with
OpenWorkload	OpenWorkloadUser
OpenWorkloadUser	CompositeAction
ClosedWorkload	ClosedWorkloadUser
ClosedWorkloadUser	CompositeAction
SimpleAction	Any Action, ActiveResource
CompositeAction	Any Action
ForkAction	Any Action
JoinAction	Any Action
AcquireAction	Any Action, PassiveResource
ReleaseAction	Any Action, PassiveResource
ActiveResource	SimpleAction
PassiveResource	AcquireAction, ReleaseAction

Table 1: The different kinds of simulation processes used to define the process-oriented simulation model

OpenWorkloadUser, except that after the activated use case terminates, it waits a random amount of time and then activates another use case.

SimpleAction Represents a computation performed on the system. The computation may be repeated a number of time, and may request service from an active resource. It interacts first with the associated active resource, and then with one of the successor actions.

CompositeAction Models a composite computation, made of a number of sub-computations. Its behavior consists on activating the first (root) substep.

ForkAction Represents the start of a parallel execution of two concurrent computations. Its actions consists on activating all the successor actions.

JoinAction Represents the end of a parallel computation. This process waits for all the predecessor actions to complete, and then activates one of its successors.

SimAcquireAction Represents an action requiring a passive resource. It interacts with the passive resource before activating one of the successor actions.

SimReleaseAction Represents an action releasing a passive resource. It interacts with the passive resource before activating one of the successor actions.

ActiveResource This process represents an active resource. It waits for requests, satisfying them according to its scheduling policy.

PassiveResource This process represents a passive resource. It waits for requests, checking wether they can be satisfied; thus, it interacts with processes of class simAcquireAction and SimReleaseAction.

Fig. 8 illustrates the structure of the simulation model, that is similar to that just seen for the UML model representation. The root class of the hierarchy is simObject, which is derived from the process class of the `libcppsim` library. The three classes simActionBase, simWorkload and simResource respectively represent actions, workloads and resources, and they all inherit from the base simObject class.

The class hierarchy of the simulation actions shown in Fig. 8, follows a pattern similar to that of the corresponding UML model representation illustrated in Fig. 6. The root class is simActionBase. Its

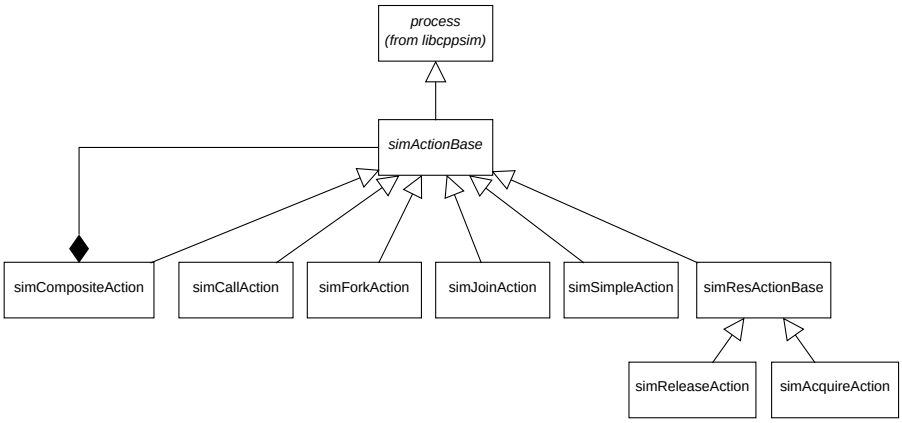


Figure 8: Class diagram for UML-Ψ simulation actions

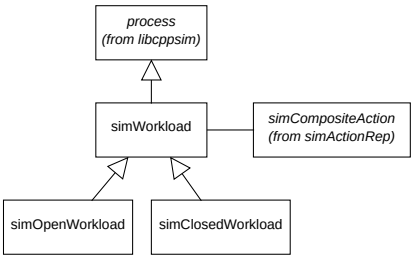
dynamic behavior is not specified; this allows subclasses to implement specific behaviors, according to their type. Various subclasses implement different types of actions: atomic actions (simSimpleAction), composite actions (simCompositeAction), fork and join nodes (simForkAction and simJoinAction respectively), actions dealing with passive resources (simResActionBase), and the special call action (simCallAction). A call action is needed to overcome a limitation of ArgoUML and Poseidon, which at the moment do not provide functionalities to define composite actions, even if they are defined in the UML standard. As a workaround, a composite action is modeled as follows. The user must specify a Use Case associated with the content of the composite action. This is necessary as ArgoUML/Poseidon do not allow users to create an activity diagram without it being used to specify a class or Use Case behavior. At this point, a simple (atomic) action is used where the composite action would have been put. This atomic action is stereotyped as ≪ PAcompositeStep ≫. A tagged value labeled `ActivateUC` must contain the name of the Use Case to activate when the action is to be executed.

Fig. 9 shows the class diagram representing simulation workloads and resources. The simWorkload virtual base class represents a generic workload, whereas specialized simOpenWorkload and simClosedWorkload classes are used to model open and closed simulation workload processes, respectively.

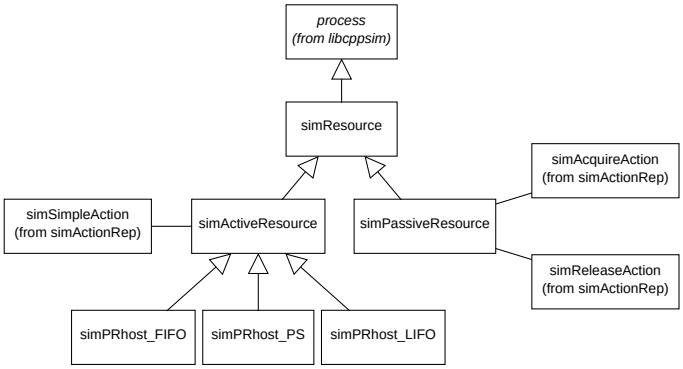
The simResource virtual base class is the root of the resource processes hierarchy. Resources are specialized in active resources, or processors (simActiveResource) and passive resources (simPassiveResource). Active resources are specialized again depending on their scheduling policy. Hence, we define a class representing a simulation process of a processor with FIFO scheduling policy (simPRHost_FIFO), one for the LIFO scheduling policy (simPRHost_LIFO) and one for the Processor Sharing (PS) scheduling policy (simPRHost_PS). Adding more scheduling policies to the simulation engine requires the user to derive a suitable class from simActiveResource.

2.3 Simulation execution

Once the performance model is set up, UML-Ψ starts the simulation by activating the simulation processes representing the workloads and the resources. The workload processes create the requests arriving to the system, which in turn will activate the corresponding activity diagrams. The simulation engine is based on `libcppsim`, a portable C++ library for process-oriented simulation described in [11]. It provides classes for representing simulation processes, and facilities for output data analysis and random



(a) Workloads



(b) Resources

Figure 9: Class diagram for UML-Ψ simulation workloads and resources

variate generation.

The simulation computes the following figures of merit:

- utilization of the resources;
- throughput of the resources;
- mean execution time of Activity diagrams associated to each Use Case.

All simulation results refer to the steady-state behavior of the system. Due to the statistical nature of simulation results, and given that simulation executes only for a finite amount of time, it is very important to apply an appropriate output analysis technique. The `libcppsim` library implements the batch means method for computing the mean of a sequence of observation, after deleting an initial prefix of that sequence for removing the initialization bias [6, 9]; we discard by default the first 20% of each sequence of observations. The method of independent replications [9] is also being implemented.

Simulation results are computed with their confidence intervals, i.e., the user can specify the confidence level and the desired accuracy. We assume by default a 90% confidence level and a relative error on the simulation results of 10%. We consider the accuracy as the relative error on the results, i.e., the ratio between the interval width and its central value.

The simulation is executed until one of the following conditions hold:

1. all the performance results have been computed with the requested accuracy,

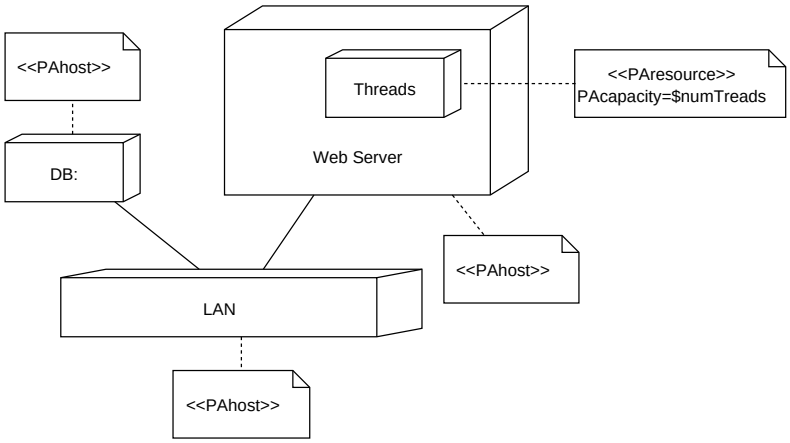


Figure 10: Deployment diagram for the two-tier E-Commerce site

2. the simulated time exceeded a user-defined threshold (the default value is 10^6 simulation time units),
3. the number of event notices processed by the simulator exceed a user-defined threshold (the default value is 2^{24} event notices).

If the simulation does not converge after the maximum simulation duration has been reached, the user is asked if the simulation run has to be extended. In this way the user can obtain results even for simulation models which take longer to converge.

2.4 Feedback

After the simulation experiment has been completed, the simulation results are inserted into the UML model as tagged values associated to the relevant elements. These tags are named `PAutilization` and `PAthroughput` for the utilization and throughput of active and passive resources respectively, and `PArespTime` for the mean execution time of actions and whole Use Cases. In this way, interpreting the simulation results is easy. This makes out approach different from most of the existing ones. In approaches where the performance model is structurally different from the software model, interpreting the performance measures at the software level can be less immediate. The reason is that elements of the performance model, where results are computed, may have no direct correspondence to software elements, so that the tool must combine different measures to derive software metrics.

3 Case Study

In this section we describe with a case study how UML-Ψ works. We consider the architecture of a two-tier E-Commerce site. The Deployment diagram of Fig. 10 describes the hardware resources available in the system. The system consists of a Database server and a Web Server connected through a LAN. The Web Server is multithreaded, with a maximum number of threads. Before any user request can be processed, a thread must be acquired. If all threads are busy, user requests are queued in FIFO order until one becomes available. The LAN, Database server and Web server are modeled as active resources (processors). The pool of web server threads is modeled as a passive resources with capacity equal to the maximum number of threads.

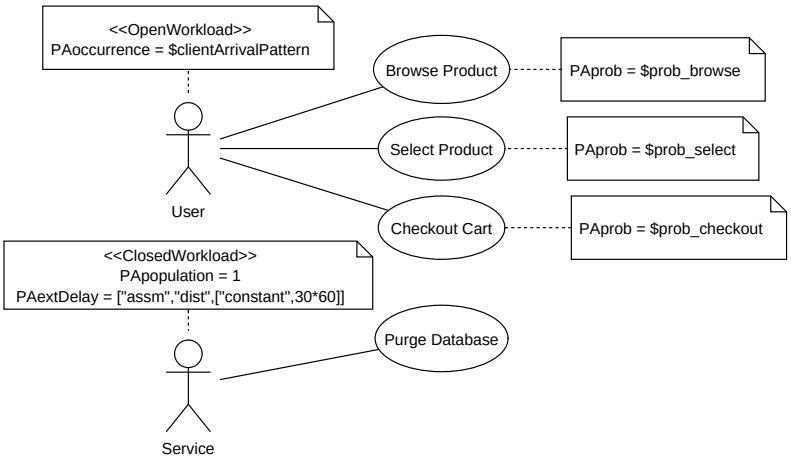


Figure 11: Use Case diagram for the E-Commerce site

Note that the value of the `PAcapacity` tag associated to the Threads node is a Perl variable. UML-Ψ uses the Perl language to parse the values of tags, so it is possible use arbitrarily complex Perl expressions. It is also possible (as in this case) to use unbounded variables. When the simulation program is executed, the user must supply a definition file which is parsed by the Perl interpreter before evaluating tag values. The definition file is expected to set a value for all unbounded variables occurring in the UML model. The name of the definition file is given as the value of the `paramFileName` tag associated to the whole UML model.

The usage scenario of the E-Commerce system is described by the Use Case diagram of Fig. 11. We identify two user classes, represented by the Actors named "User" and "Service" respectively. The "User" Actor represent the sequence of requests arriving at the E-Commerce Web server from external users. The inter-arrival time between request is equal to `$clientArrivalPattern`. User requests are of three different kinds, which result in three different interactions with the system. Users may browse one product from the catalog, with probability `$prob_browse`, add one product to their web cart with probability `$prob_select` and confirm the order (purchase all the products in their cart) with probability `$prob_checkout`. All these probabilities must sum to 1.

The behavior of the system can be specified in detail by associating an Activity diagram to each use case of Fig. 11 and 13 through 16 describe the interactions associated to the *Browse Product*, *Purge Database*, *Select Product* and *Checkout cart* use cases, respectively. Annotations are described inside UML notes for convenience only. UML editors usually do not display tagged values in UML models in order to avoid visual clutter. Fig. 12 shows a screenshot of Poseidon [17] while designing the software model.

Each action state must be stereotyped using exactly one of the following stereotypes:

≪PAstep≫ to denote an action requiring service from an active resource. We can specify the service demand, the number of times the request is repeated and the time between repetitions.

≪GRMacquire≫ to denote an action requiring one or more instances of a passive resource. The request blocks until the specified amount of passive resource is available. Multiple requests queued at the same resource are serviced in FIFO order.

≪GRMrelease≫ to denote an action releasing one or more instances of a passive resources.

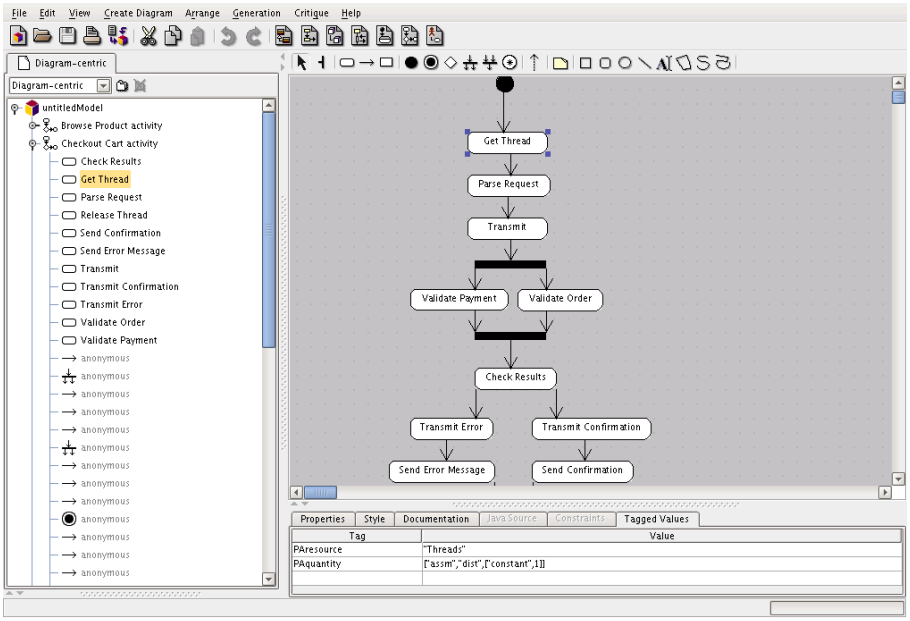


Figure 12: Annotating UML specifications with Poseidon [17]

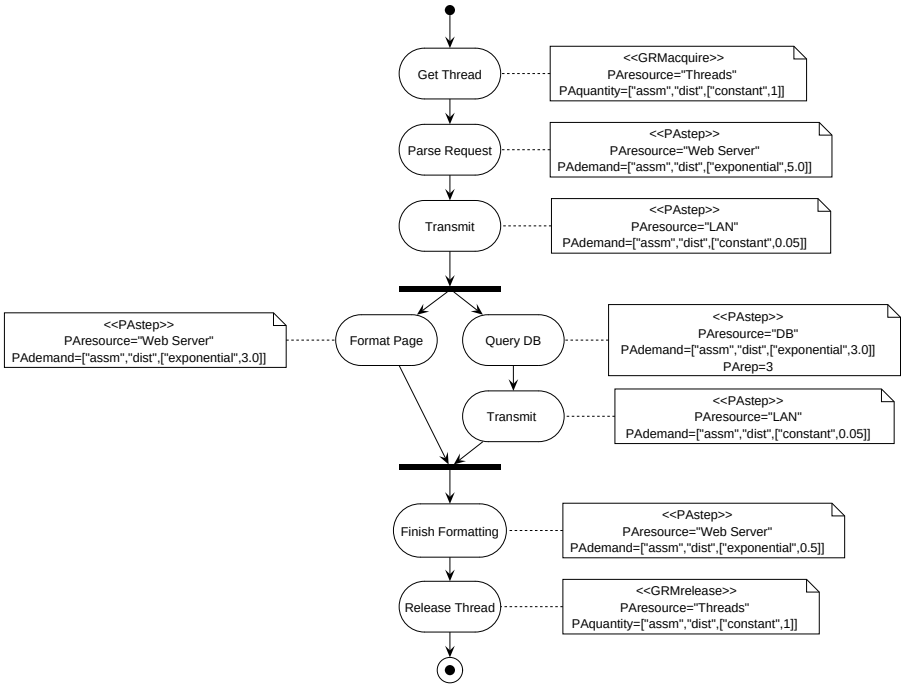


Figure 13: Activity diagram corresponding to the *Browse Product* use case

A complete description of the stereotypes and tagged values used to annotate the UML model can be found in [5, 12].

As can be seen in the figures, there are simulation parameters which are left unspecified. These variables can be defined in an external Perl file, which is parsed by UML-Ψ before building the simulation model. In this example, we consider a configuration file with the following content:

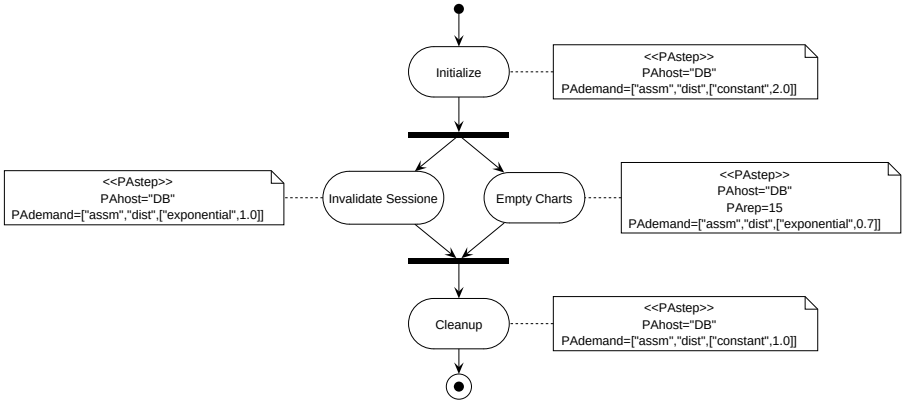


Figure 14: Activity diagram corresponding to the *Purge Database* use case

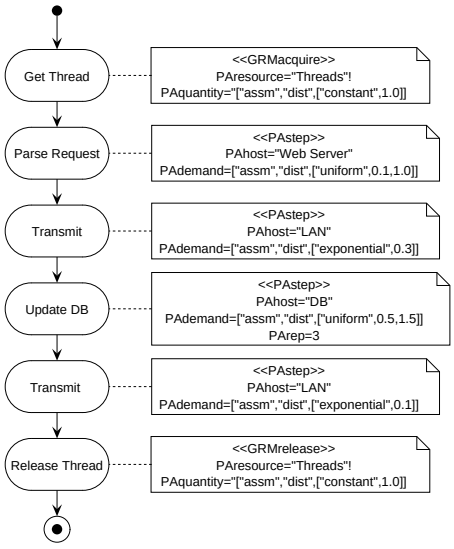


Figure 15: Activity diagram corresponding to the *Select Product* use case

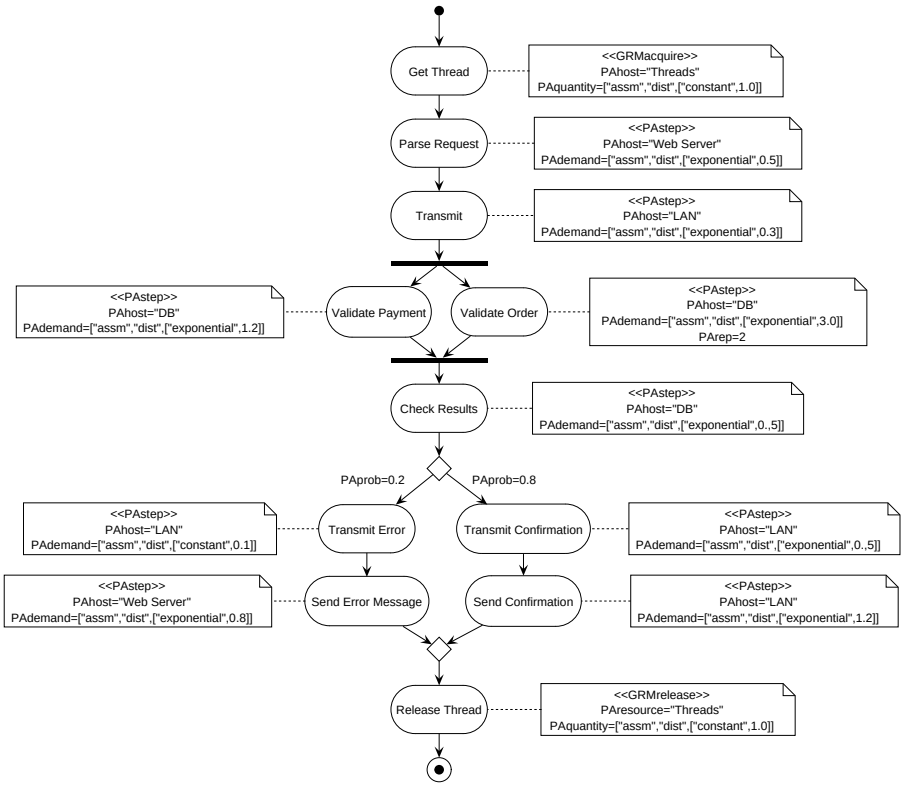


Figure 16: Activity diagram corresponding to the *Checkout cart* use case

```
$prob_browse=0.8;
$prob_select=0.1;
$prob_checkout=0.1;
$clientArrivalPattern=["unbounded",["exponential",60.0]];
$num_threads=5;
$simduration=4000000;
$confrelwidth=0.1;
```

The last two parameters in the configuration file set the maximum simulation duration to 4×10^6 simulated time units, and the accuracy to 0.1, that is the width of confidence intervals must be at most 10% of their central values.

Simulation results are summarized in Table 2. The results show that the utilization of the Web and Database servers are quite low (about 15% and 11% respectively). However, the mean execution time of the Browse Product activity is greater than the time required to confirm an order. The time needed to browse the catalog is very high (about $17s$) and is clearly unacceptable for interactive use. The software modeler should use these informations to modify the structure of the software system in order to improve the situation. UML-Ψ also computes the mean execution time of each action, so the software engineer can use this information to check where the most time is spent while executing the Activity diagram of Fig. 13.

	Lower bound	Upper bound
DB Utilization	0.150885	0.150928
DB Throughput	0.0657735	0.0657854
Web Server Utilization	0.118278	0.118315
Web Server Throughput	0.0454527	0.0454688
LAN Utilization	0.0030746	0.00307834
LAN Throughput	0.0319812	0.0319959
Threads Utilization	0.254308	0.254368
Threads Throughput	0.0168095	0.0168234
Threads WaitingTime	0.0011884	0.00531334
Browse Product Resp. Time	17.4728	17.8715
Select Product Resp. Time	5.93638	6.38555
Purge database Resp. Time	24.7636	26.8713
Checkout Resp. Time	12.6457	12.7854

Table 2: Simulation results

4 Conclusions

In this paper we have presented UML-Ψ, a simulation-based performance evaluation tool for early assessment of software performances. UML-Ψ transforms annotated UML diagrams into a simulation performance model, implements the model using process-oriented simulation and evaluates the performance model. Simulation results are computed with steady state analysis and with confidence intervals, and are inserted back into the UML model as new tagged values associated to the relevant model elements. The UML-Ψ tool is one of the first software performance evaluation tools which is based on the UML Performance Profile [13]. The tool aims at being easy to use even for users without specific training in simulation. Being implemented in C++ makes it efficient and easily portable to different Operating Systems.

Currently we are extending UML-Ψ in several directions. First, we are planning to use a larger subset of the annotations from [13], in order to allow the modeler to describe the software in more detail using different kinds of UML diagrams. We are also trying to integrate the UML-Ψ tool with other software performance approaches based on different performance models derived from the same UML model based on different modeling notations, e.g., Queuing Networks and Stochastic Process Algebras. The ultimate goal is to integrate different kinds of quantitative software analysis techniques into a general framework allowing different kinds of quantitative and qualitative analyses, e.g., reliability [10], on the same software specification. Finally, we are evaluating how the UML-Ψ tool, and the associated software performance evaluation approach, can be extended to cope with the forthcoming UML 2.0. The new version of UML is formally defined. This is very useful for validating the simulation model, that is, proving that the simulation model can actually be substituted to the software system for performance evaluation purposes.

Acknowledgments

This work has been partially supported by MIUR research project FIRB “Performance Evaluation of Complex Systems: Techniques, Methodologies and Tools”.

References

[1] D. Al-Dabass, editor. *Proc. of ESM'03, the 17th European Simulation Multiconference*, Nottingham, UK, June 9–1 2003. SCS–European Publishing House.

[2] ArgoUML – Object-oriented design tool with cognitive support. http://www.argouml.org/.

[3] L. B. Arief and N. A. Speirs. A UML tool for an automatic generation of simulation programs. In Proceedings of WOSP 2000 [18], pages 71–76.

[4] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5), May 2004. To appear.

[5] S. Balsamo and M. Marzolla. Simulation modeling of UML software architectures. In Al-Dabass [1], pages 562–567.

[6] J. Banks, editor. *Handbook of Simulation*. Wiley–Interscience, 1998.

[7] M. De Miguel, T. Lambolais, M. Hannouz, S. Betgé-Brezetz, and S. Piekarec. UML extensions for the specifications and evaluation of latency constraints in architectural models. In Proceedings of WOSP 2000 [18], pages 83–88.

[8] A. Hennig, D. Revill, and M. Ponitsch. From UML to performance measures – simulative performance predictions of IT-systems using the Jboss application server with OMNET++. In Al-Dabass [1].

[9] K. Kant. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, Int. Editions, 1992.

[10] P. Katsaros and C. Lazos. A simulation test-bed for the design of dependable e-services. *WSEAS Transactions on Computers*, 4(2):915–919, 2003.

[11] M. Marzolla. `libcppsim`: a Simula-like, portable process-oriented simulation library in C++. Tech. Report CS-2004-1, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy, Feb. 2004. To appear in Proceedings of ESM'04.

[12] M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD Thesis TD-2004-1, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy, Feb. 2004.

[13] Object Management Group (OMG). UML profile for schedulability, performance and time specification. Final Adopted Specification ptc/02-03-02, OMG, Mar. 2002.

[14] Object Management Group (OMG). XML Metadata Interchange (XMI) specification, version 1.2, Jan. 2002.

[15] I. Opnet Technologies. Opnet modeler. http://www.opnet.com.

[16] Perl Home Page. http://www.perl.com.

[17] Poseidon for UML. http://www.gentleware.com/.

[18] *Proc. of the Second International Workshop on Software and Performance (WOSP 2000)*, Ottawa, Canada, Sept. 2000. ACM Press.

[19] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[20] C. U. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[21] Softeam. Objecteering CASE tool. http://www.objecteering.com.

[22] A. Varga. The OMNET++ discrete event simulation system. In *Proc. of ESM'01, the 15th European Simulation Multiconfer ence*, pages 319–324, Prague, Czech Republic, June 6–9 2001.

[23] L. Wall, T. Christiansen, and J. Orwan. *Programming Perl*. O'Reilly & Associates, third edition, July 2000.