Software Model-based Performance Analysis

Dorina C. Petriu

Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, Canada, K1B4G2
petriu@sce.carleton.ca

ABSTRACT. *The chapter starts with a brief review of performance modelling formalisms and a discussion of the performance annotations that need to be added to UML software models in order to enable performance analysis. The principles for transforming annotated software models into performance models are presented next. Such model transformations must bridge a large semantic gap between the source and the target model; hence a pivot model is often used. An example of such a transformation is given, from UML extended with the MARTE profile to the Layered Queueing Network performance model. The role of an intermediate pivot language named Core Scenario Model is also discussed. The chapter ends with a discussion of lesson learned and future challenges for integrating the analysis of multiple non-functional properties in the context of MDE.*

KEYWORDS: *performance analysis, model transformation, UML, MARTE, LQN.*

1. Introduction

Model-Driven Development (MDD) is an evolutionary step in the software field that changes the focus of software development from code to models. MDD is based on *abstraction* to separate the model of the application under construction from underlying platform models, and *automation* to generate code from models. The emphasis on models facilitates the analysis of non-functional properties (NFP), such as performance, scalability, reliability, security, safety, etc. of the software under development based on its model. This brings more "engineering" into software development, leading to the paradigm known as Model-Driven Engineering (MDE).

Over the years, many formalisms and tools for the analysis of different NFPs have been developed, for example queueing networks, stochastic Petri nets, stochastic process algebras, fault trees, probabilistic time automata, formal logic, etc. The research challenge is to bridge the gap between MDE and existing NFP analysis formalisms and tools rather than to "reinvent the wheel". An approach for the analysis of different NFPs composed of the following steps is emerging in the literature: a) add annotations describing the respective NFP to the software model, b) define a model transformation from the annotated software models to the formalism used for NFP analysis, c) analyze the NFP model using existing solvers, and d) give feedback to designers. Figure 1 illustrates this process for the case where performance is the analyzed NFP; similar "analysis loops" exist for other NFPs.

In the case of UML-based software development, the extensions required for NFP-specific annotations are defined as UML profiles, which provide de additional advantage to be processed by standard UML tools without any change in the tool support. Two standard UML profiles provide, among other features, the ability to define performance annotations: the *UML Profile for Schedulability, Performance and Time* (SPT) defined for UML 1.X versions (OMG, 2005) and the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) defined for UML2.X versions (OMG, 2009).

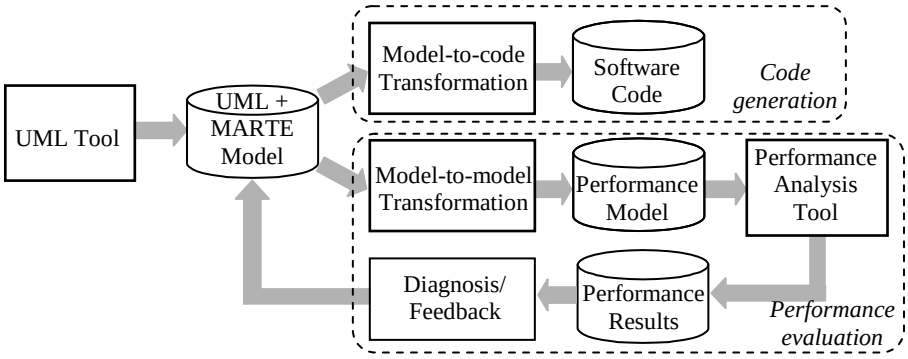


Figure 1. *MDE model transformations for code generation and performance analysis*

Software Performance Engineering (SPE) is a methodology introduced by (Smith, 1990) with the aim to insure that software products are built to meet their performance requirements. SPE uses predictive performance models to evaluate the temporal responsiveness of the system (such as response times, delays and throughputs) and to compare architectural and design and alternatives for systems with timing and capacity requirements. SPE begins early in the software lifecycle, before serious barriers to performance are frozen into the design and implementation. Since the introduction of SPE, there has been a significant effort to integrate performance analysis into the software development process throughout all lifecycle phases. A good survey of the techniques for deriving performance models from software specifications is given in (Balsamo *et al.*, 2004).

In the traditional SPE, performance models are built by hand by the analyst. The emergence of model-driven engineering has triggered research in the automatic transformation of software models into performance models. A high degree of automation in building performance models and interpreting their results brings the following benefits:

- higher consistency between the design specification and performance model;
- traceability of performance effects back to design elements and decisions;
- fast performance model update and re-evaluation after a design change;
- stronger analysis for recommending design changes;
- means to bridge the gap between the designer and the performance engineer.

2. Performance Models

There are two kinds of approaches for analyzing the timing properties of real-time systems: performance and schedulability analysis. Their purpose is different: the first is concerned with estimating average resource capacity, queueing delays due to contention for resources, throughput and identifying bottlenecks, while the second is concerned with finding a feasible schedule that guarantees deadlines inherent to hard real-time systems.

Performance analysis is applied to best-effort and soft real-time systems, such as information processing systems, web-based applications and services, multimedia, telecommunications, enterprise systems. The input parameters and performance results are stochastic variables/processes. On the other hand, schedulability analysis is applied to hard real-time systems with strict deadlines, such as embedded systems, and the analysis is often based on worst-case execution time and deterministic assumptions. This chapter focuses on performance analysis.

It is worth observing that both performance and schedulability models represents the system at runtime, so the models must include not only the performance characteristics of the software itself, but also of the underlying platforms (operating system, middleware, hardware).

A performance models is an abstract representation of a real system that captures its performance properties – mostly related to the quantitative use of resources

during runtime behaviour – and is capable of reproducing its performance. The model can be used to study the performance impact of different design and/or configuration alternatives under different workloads, leading to advice for improving the system. Performance evaluation of a model may be done either by solving a set of equations by some analytical (possibly numerical) methods or by simulating the model and collecting statistical results. Analytical performance models are usually based on underlying stochastic models, which are often assumed to be Markov processes. A Markov process is a stochastic process with discrete state space, where all information about the future evolution of the process is contained in the present state, and not on the path followed to reach this state. Markov models suffer from a problem known as *state space explosion*, whereby its number of states grows combinatorially with the performance model size. This may introduce severe limitations in the size of performance models that can be solved.

Examples of well-known analytical performance models are queueing networks, stochastic Petri nets, stochastic automata networks and stochastic process algebra.

Queueing Network (QN), one of the best known performance models captures very well the contention for resources (Lazowska et al., 1984). Efficient analytical solutions exist for a class of QN (*separable* or *product-form* QN), which make it possible to derive steady-state performance measures without resorting to building the underlying state space. The advantage is that the solution is faster and larger models can be solved. The disadvantage consists in restrictions on model assumptions (e.g., service time distributions, arrival process, scheduling policies). Similar to the approach for product-form QN, approximate solutions have been developed for non-separable QN. There are many extensions to QN in literature. One of them, Layered Queueing Networks, will be discussed in section 2.2.

Stochastic Petri Nets (SPN) (Ajmone Marsan et al., 1995) are very good flow models able to represent concurrency, but not as good at representing resource contention and especially queueing policies. Efficient solutions exist only for a limited class of SPN; most interesting models are solved with Markov chain-based solutions.

Stochastic Automata Networks (Plateau *et al*., 1991) are composed of modular communicating automata synchronized by shared events and executing actions with random execution times. The main disadvantage is the state space explosion of its Markovian solution.

Stochastic Process Algebra, introduced in (Hillston, 1994), takes a compositional approach by decomposing the system into smaller subsystems easier to model. This approach is based on an enhanced process algebra, Performance Evaluation Process Algebra (PEPA). The compositional nature of the language provides benefits for model solution as well as model construction. The solution is based on the underlying Markov process.

The target performance model in this chapter is a QN extension called Layered Queueing Network (LQN). The following discussion focuses on QN and LQN.

2.1. Queueing Network Models

A Queueing Network (QN) model is a directed graph, whose nodes are service centres, each representing a resource in the system and arcs with associated routing probabilities (or visit ratios) determine the paths that customers take through the network. Customers representing jobs are flowing through the system, competing for resources. QN are used to model systems with stochastic characteristics. A QN may have more than one customer class. Each class contains statistical identical customers and has its own workload intensity, service demands and visit ratios. The workload of a customer class may be open (customers arrive with a certain rate, spend time in the system being served, then leave the system) or closed (the number of customer is fixed; after completing a cycle, a customer starts again). The performance results will be obtained by customer class.

A single service center containing a server and a queue has the following characteristics (represented by Kendall's notation *A/S/c/m/N*):

A = arrival process (e.g., M-Markov, G-general, D-deterministic distribution)
S = service rate (uses distribution identifiers as above)
c = number of servers available serve the customers from the queue
m = capacity of the queue (infinite by default)
N = customer population (also infinite by default)
scheduling policy (FIFO, LIFO, PS, preemptive priority, etc.).

An important characteristic of QN models is that the functions expressing the queue length and waiting time at a server with respect to workload intensity are very non-linear. An intuitive explanation is as follows: at low workload intensity, an arriving customer meets low competition, so its residence time is roughly equal to its service demand; as the workload intensity rises, congestion increases, and the residence time along with it; as the service center approaches saturation, small increases in arrival rate result in dramatic increases in residence time (Lazowska *et al.*, 1984). The non-linearity of performance results makes it difficult to estimate the system performance by simple "rules of thumb", without solving the system of non-linear equations with QN solvers.

Another important concept in a QN is the bottleneck service center, the one that saturates first and throttles the system. Identifying correctly the bottleneck center is important for performance analysis, because the bottleneck must be relieved first in order to improve the system performance. In the case of multiple customer classes, the bottleneck may be different for each class.

QN are widely used for modeling a variety of systems. Although they represent a system at a rather abstract level, QN are a useful tool for predicting the performance of a system. The expected accuracy of QN models according to experience is within 5% to 10% for utilizations and throughputs and within 10% to 30% for response times (Lazowska *et al*, 1984).

2.2. Layered Queueing Network Model

The LQN model (Woodside et al., 1995) is a QN extension which can represent nested services (i.e., a server may also be also a client to other servers). A LQN model is a graph whose nodes are either software tasks (thick rectangles) or hardware devices (circles) and the arcs denote service requests, as illustrated in Figure 2. The nodes with outgoing but no incoming arcs play the role of clients, the intermediate nodes with both incoming and outgoing arcs are usually software servers and the leaf nodes are hardware servers. A software or hardware server node can be either a single-server or a multi-server. Software tasks have entries corresponding to different services. Although not explicitly shown in the LQN notation, every server (software or hardware) has an implicit message queue where incoming requests for any offered service are waiting their turn. There are three types of service requests: synchronous (filled arrow), asynchronous (stick arrow) and forwarding (dotted arrow).

Figure 2 shows an example of an LQN model of a web server: at the top there are two customer classes with a given number of stochastically identical clients. Each client sends demands for two services e3 and e4 of the WebServer (drawn as thin rectangles attached to the respective task). Every entry has its own execution times and demands for other services (given as model parameters). In this case, the WebServer entries require a service from eCommServer, which in turn calls different entries of two database tasks – a secure and a regular one. Each software task is running on a processor shown as a circle. Also as circles are shown the communication network delays and the disks used by the databases.

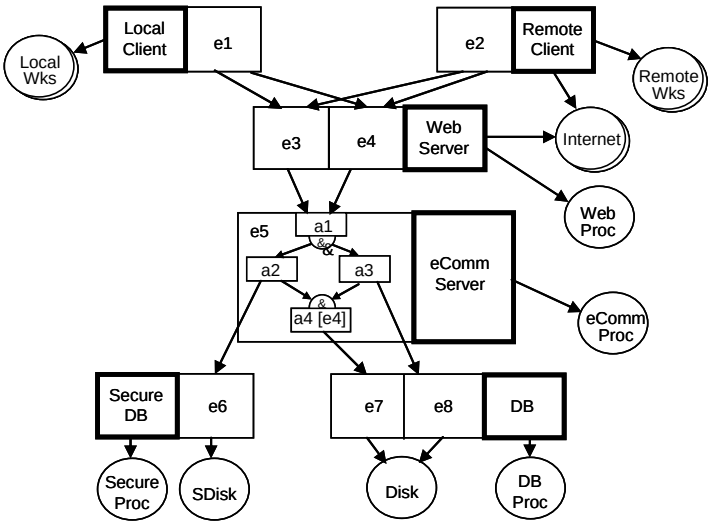


Figure 2.*Example of LQN model*

All arcs used in this example represent synchronous requests, for which the sender is blocked until it receives a reply from the provider of service. It is possible to have also asynchronous requests, where the request sender does not expect any reply from the server. Another communication style in LQN called “forwarding”, allows for a client request to be processed by a chain of servers: the first server in the chain will forward the request to the second, etc., and the last server in the chain will reply to the client.

A server entry may be decomposed in two or more sequential phases of service. Phase 1 is the portion of service when the client is blocked waiting for a reply from the server (it is assumed that the client has made a synchronous request). At the end of phase 1, the server will reply to the client, which will unblock and continue its execution. The remaining phases, if any, will be executed in parallel with the client. An extension to LQN (Franks, 2000) allows for an entry to be decomposed into activities if more details are required to describe its execution (as for example entry *e5* of task *eCommServer* in Figure 2). The activities are connected together to form a directed graph, which may branch into parallel threads of control like in Figure 2, or may choose randomly between different branches. Just like phases, activities have execution time demands, and can make service requests to other entries.

3. Software Model with Performance Annotations

3.1. *Performance domain model*

In order to understand what kind of performance annotations need to be added to UML software models, we need to look at the basic concepts – or in other words at the domain model – for performance analysis. Performance is determined by how the system behaviour uses system resources. Scenarios define execution paths with externally visible end points. Quality of Service (QoS) requirements (such as response time, throughput, probability of meeting deadlines, etc.) can be placed on scenarios. In SPT, the performance domain model describes three main types of concepts: resources, scenarios, and workloads (OMG, 2005).

The resources used by the software can be active or passive, logical or physical software or hardware. Some of these resources belong to the software itself (e.g., critical section, software server, lock, buffer), others to the underlying platforms (e.g., process, thread, processor, disk, communication network).

Each scenario is composed from scenario steps joined by predecessor-successor relationships, which may include fork/join, branch/merge and loops. A step may represent an elementary operation or a whole sub-scenario. Quantitative resource demands for each step must be given in the performance annotations. Each scenario is executed by a workload, which may be open (i.e., requests arriving in some predetermined pattern) or closed (a given number of users or jobs).

In the SPT profile, the domain models for schedulability and performance and their corresponding sub-profiles were defined independently, which made it difficult to reuse annotated models for different analyses. In MARTE (OMG, 2009), the foundation concepts and non-functional properties (NFPs) shared by different quantitative analysis domains are joined in a single package called Generic Quantitative Analysis Model (GQAM), which is further specialized by the domain models for schedulability (SAM) and performance (PAM). Other domains for quantitative analyses, such as reliability, availability, safety, are currently being defined by specializing GQAM.

Core GQAM concepts describe how the system behavior uses resources over time, and contains the same three main categories of concepts presented at the beginning of the section: resources, behaviour and workloads.

GQAM Resource Concepts. A resource is based on the abstract *Resource* class defined in the General Resource Model and contains common features such as scheduling discipline, multiplicity, services. The following types of resources are important in GQAM: a) *ExecutionHost*: a processor or other computing device on which are running processes; b) *CommunicationsHost*: hardware link between devices; c) *SchedulableResource*: a software resource managed by the operating system, like a process or thread pool; and d) *CommunicationChannel*: a middleware or protocol layer that conveys messages.

Services are provided by resources and by subsystems. A subsystem service associated with an interface operation provided by a component may be identified as a *RequestedService*, which is in turn a subtype of *Step*, and may be refined by a *BehaviorScenario*.

GQAM Behaviour/Scenario Concepts. The class *BehaviorScenario* describes a behavior triggered by an event, composed of *Steps* related by predecessor-successor relationships. A specialized step, *CommunicationStep*, defines the conveyance of a message. Resource usage is attached to behaviour in different ways: a) a Step implicitly uses a *SchedulableResource* (process, thread or task); b) each primitive Step executes on a host processor; c) specialized steps, *AcquireStep* or *ReleaseStep*, explicitly acquire or release a *Resource*; and d) *BehaviorScenarios* and *Steps* may use other kind of resources, so *BehaviorScenario* inherits from *ResourceUsage* which links resources with concrete usage demands.

GQAM Workload Concepts. Different workloads correspond to different operating modes, such as takeoff, in-flight and landing of an aircraft or peak-load and average-load of an enterprise application. A workload is represented by a stream of triggering events, *WorkloadEvent*, generated in one of the following ways: a) by a timed event (e.g. a periodic stream with jitter); b) by a given arrival pattern (periodic, aperiodic, sporadic, burst, irregular, open, closed); c) by a generating mechanism named *WorkloadGenerator*; d) from a trace of events stored in a file.

As mentioned above, the Performance Analysis Model (PAM) specializes the GQAM domain model. It is important to mention that only a few new concepts were defined in PAM, while most of the concepts are reused from GQAM.

PAM specializes a *Step* to include more kinds of operation demands during a step. For instance, it allows for a non-synchronizing parallel operation, which is forked but never joins (*noSync* property). A new step subtype, *PassResource*, indicates the passing of a shared resource from one process to another.

In term of Resources, PAM reuses *ExecutionHo*st for processor, *Schedulable Resources* for processes (or threads) and adds a *LogicalResource* defined by the software (such as semaphore, lock, buffer pool, critical section). A runtime object instance (*PaRunTInstance*) is an alias for a process or thread pool identified in behavior specifications by other entities (such as lifelines and swimlanes).

A UML model intended for performance analysis should contain a structural view representing the software architecture at the granularity level of concurrent runtime components and their allocation to hardware resources, as well as a behavioural view showing representative scenarios with their respective resource usage and workloads.

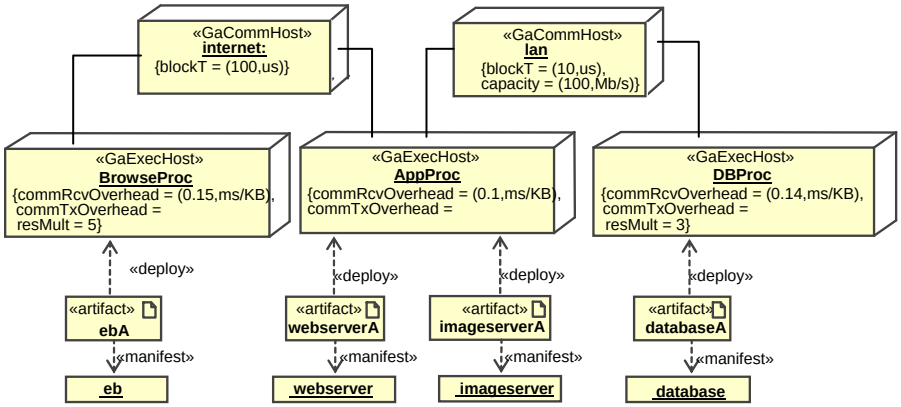
3.2. *Source model example*

This section presents an example of UML+MARTE source model based on TPC-W, a benchmark of the Transaction Processing Performance Council which models the workload of an on-line bookstore (TPC, 2002).

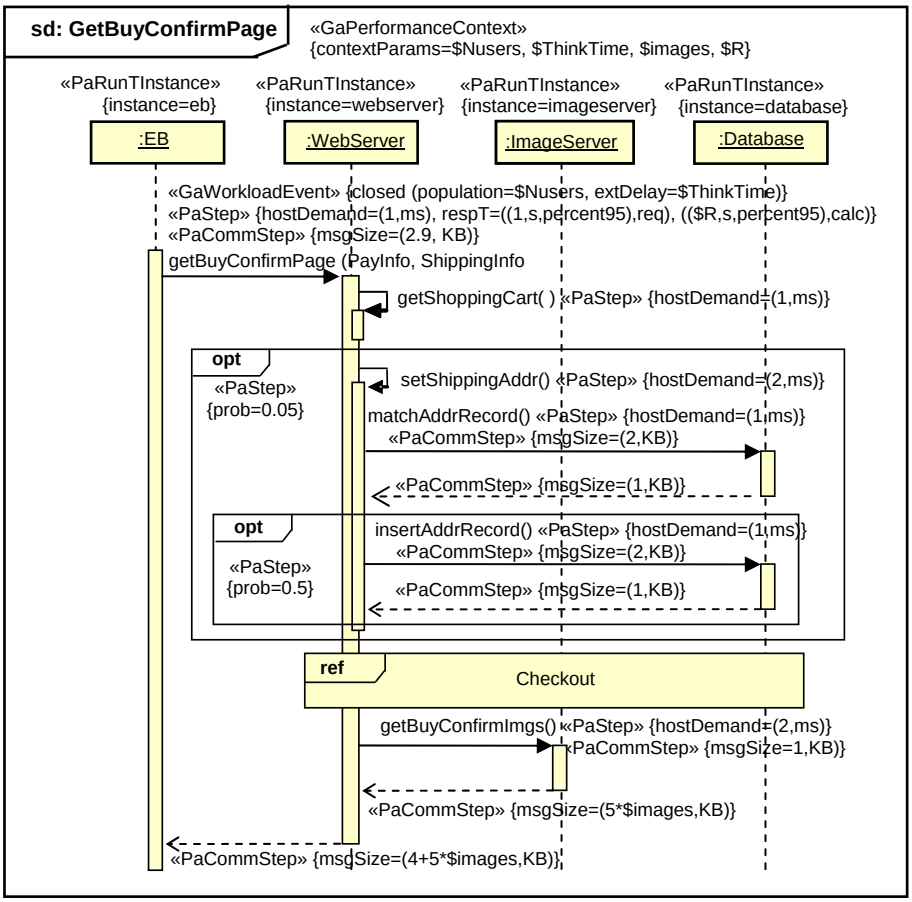
The components of TPC-W are logically divided into three tiers: a) a set of emulated web browsers (EB), b) a web tier including web servers and image servers and c) a persistent storage tier. TPC-W emulates customers browsing and buying products from a website, with 14 different web pages that correspond to typical customer operations. The user starts at the “Home” page that includes the company logo, promotional items and navigation options to best selling books, new books, search pages, the shopping cart, and order status pages. At every page, the user is offered a selection of pages that can be visited next. The user may browse pages containing product information, perform searches with different keys and put items in the cart.

A new customer has to fill out a customer registration page; for returning customers, the personal information is retrieved from the database. Before ordering, the user may update the shopping cart content. When deciding to buy, the user enters the credit card information and submits the order. The system obtains credit card authorization from a Payment Gateway Emulator (PGE) and presents the user with an order confirmation page. At a later date the user can view the status of the last order.

The UML+MARTE source model to be transformed in a performance model is shown in Figure 3. It is composed of a structural view showing the concurrent runtime component instances and their deployment to processors in Figure 3.a, and a behavioral view showing the scenario for one of the pages needed for buying pro-



a) *Deployment diagram*



b) *GetBuyConfirmPage scenario*

Figure 3. *Target UML model with MARTE performance annotations*

ducts in Figure 3.b. Usually the source model contains several performance-critical scenarios that are used to generate the system performance model, but only one is given here due to space limitations.

The deployment diagram from Figure 3.a shows the runtime components at the bottom, their corresponding artifacts and the deployment on processing nodes. The processing nodes are stereotyped as *«GaExecHost»* and the communication network nodes as *«GaCommHost»*. The stereotype attributes *commRcvOvh* and *commTxOvh* are host-specific costs of receiving and sending messages, *resMult=5* describes a symmetric multiprocessor with 5 processors, while *blockT* and *capacity* describe a pure latency and bandwidth for the link.

The scenario *GetBuyConfirmPage* is represented in Figure 3.b. The scenario transfers the shopping cart content into a newly created order for the registered customer, executes a payment authorization, and returns a page with the details of the order to the *EB*. The following operations are performed:

- *EB* issues a request to *WebServer* for “buy confirm page”;
- *WebServer* gets the corresponding shopping cart object;
- With 5% probability (modeled as an **opt** fragment), a shipping address is obtained and *WebServer* tries to match it with information from the database;
- If no address record is found, insert a new address record (modeled as a nested **opt** fragment)
- Invoking the *Checkout* sub-scenario (modeled as a **ref** fragment, not shown);
- *WebServer* gets necessary images from *ImageServer*;
- *WebServer* constructs the html code for “buy confirm page” and returns it to *EB*.

Some example of MARTE performance annotations used in the scenario model are used to indicate the scenario steps, the workload and the concurrent runtime instances corresponding to the lifeline roles. Two kinds of step stereotypes are applied to messages: *«PaStep»* representing the execution of the operations invoked by the message and *«PaCommStep»* for the communication costs involved with passing the message. Examples of execution step attributes are *hostDemand* giving the value and unit for the required execution time and *prob* giving the probability for the optional steps. The communication steps have an attribute *msgSize* giving the value and unit of the message size. The first step of the scenario has the scenario workload *«GaWorloadEvent»* attached to it, which defines a closed workload with a population given by the variable *$Nusers* and a think time for each user given by the variable *$ThinkTime*. Each lifeline role is related to a runtime concurrent component instance, as indicated by *«PaRunTInstance»*.

4. Mapping from Software to Performance Model

The definition of UML performance annotations has enabled research to transform UML design specifications into many kinds of performance models, based

for example on Queueing Networks (Cortellessa *et al.*, 2000), Layered Queueing Networks (Petriu *et al.*, 2002), (Petriu, 2005), (Woodside *et al.*, 2005), Stochastic Petri nets (Bernardi *et al.*, 2002), PEPA (*Cavenet et al.*, 2004), and simulation (Balsamo *et al.*, 2003).

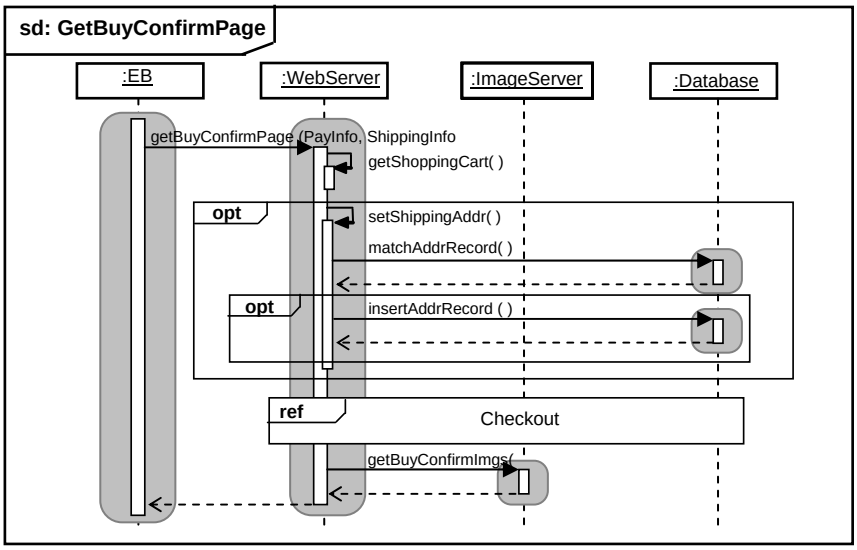
In this section, the mapping concepts from software to performance models are explained by using a direct transformation from annotated UML to LQN; another possible transformation approach using a pivot language is discussed in section 5. In the direct approach, the structure of the LQN model is generated from the high-level software architecture and deployment. In principle, active software component instances and hardware devices (which are all resources) are mapped to LQN tasks. In some cases, LQN tasks are also generated from passive instances, which are logical resources shared by active instances. In fact, the mapping to tasks is guided by the architectural patterns used in the system, such as pipeline and filters, client/server, client/broker/server, layers, master-slave, blackboard, etc. Each pattern describes two inter-related aspects: its structure (what are the interacting components) and behaviour (how they interact). The architectural patterns components are usually concurrent entities that execute in different threads of control, compete for resources, and may require some synchronization in their interaction. For more details on the transformation rules from UML to LQN based on different architectural patterns see (Petriu et al, 2000).

Figure 4 gives the direct transformation algorithm from annotated UML to LQN, assuming that the scenario models are represented by sequence diagrams. A similar pattern-based approach is presented in (Petriu *et al.*, 2002)**,** where the scenarios are modeled as activity diagrams. A graph-grammar based algorithm was proposed to divide the activity diagram into activity subgraphs, which are further mapped to LQN phases or activities. Such a transformation from software to performance model is an *abstraction-raising transformation*, as shown in (Petriu *et al.*, 2005).

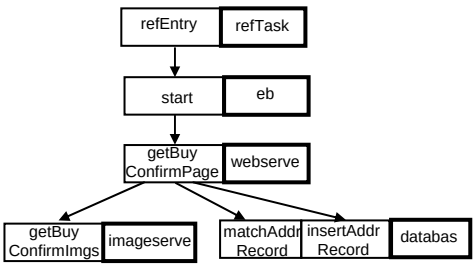
```
1. Generate LQN model structure
    1.1 map high-level component instances to LQN tasks according to patterns;
    1.2 map deployment diagram nodes to LQN hardware devices;
2. Generate LQN entries, phases, activities from scenarios
    2.1 for each scenario {
        2.1.1 generate a LQN reference task and its dummy processor corresponding
              to the scenario workload;
        2.1.1 match messages with inter-component communication style from patterns;
        2.1.2 map external message calls to entries;
        2.1.3 for each entry {
              2.1.3.1 group corresponding execution occurrences according to patterns;
              2.1.3.2 map groups to phases or activities;
              2.1.3.3 for each phase and activity
                    2.1.3.3.1 compute service time and number of calls;
              }
}
```

Figure 4. *Algorithm for direct transformation from annotated UML to LQN*

Figure 5 illustrates the application of the algorithm from Figure 4 (more specifically, the loop body 2.1.1 to 2.1.3) to the scenario *GetBuyConfirmPage* from Figure 3. It so happens that a single architectural pattern - client/server - is used repeatedly in this scenario, as the four lifeline roles interact through synchronous messages. The corresponding LQN model fragment shown in Figure 5.b contains five LQN tasks: four correspond to the active runtime component instances *eb*, *webserver*, *imageserver* and *database* (according to the *«PaRunTInstance»* stereotypes from Figure 3.b) and the fifth, *refTask*, is a reference task controlling the scenario workload. Each task has an entry for every external message it receives. For example, the database task has two entries because two different calls, *matchAddrRecord* and *insertAddrRecord* are made to this task. The shaded areas in Figure 5.a are grouping behaviour occurrences (stereotyped as scenario steps) which are further mapped to phases belonging to entries. (This example has no LQN activities). For each entry, the group of steps executed between the acceptance of the



a) *Scenario model*



b) *Corresponding LQN mode fragmentl*

Figure 5. *Mapping between the scenario model and the performance model*

corresponding synchronous request and the sending of the reply is mapped to phase 1. For instance, the steps corresponding to the behaviour executions triggered by the messages *getBuyConfirmPage*, *getShoppingCart* and *setShippingAddr* are all included in phase 1 of entry *getBuyConfirmPage*. Also included in this phase are the steps executed by this lifeline inside the fragment *Checkout* (which is not detailed here). In fact, the fragment *Checkout* may also add entries to the tasks *imageserver* and *database* corresponding to all the messages sent to these lifelines by other lifelines. The service time parameter of each phase is obtain by summing up the host demand of the included steps. Similarly is obtained the number of calls made by every phase to other entries.

5. Using a pivot language: Core Scenario Model (CSM)

A *pivot language*, also known as *intermediate* or *bridge* language, can be used as an intermediary for translation in cases where many source languages are translated to many target languages. A pivot language avoids the combinatorial explosion of translators across every combination of languages and allows for a smaller semantic gap during each transformation. Such an approach is taken, for example, in the model-driven performance evaluation project called Performance by Unified Model Analysis PUMA (Woodside *et al.*, 2005) which enables the integration of performance analysis in a UML-based software development process. PUMA uses a pivot language Core Scenario Model (CSM) to extract and audit performance information from different kinds of design models (e.g., different UML versions of activity and sequence diagrams) and to support generation of different kinds of performance models (e.g., QN, LQN, Petri nets, simulation). Figure 6 illustrates the PUMA transformation and analysis chain. There are other intermediate languages for performance analysis proposed in literature, such as Klaper (Grassi *et al.*, 2005) and Palladio Component Model (Becker *et al.*, 2007).

CSM is focused on modeling scenarios, which are implicit in many software specifications; they are useful for communicating partial behaviours among diverse stakeholders and provide the basis for defining performance characteristics.

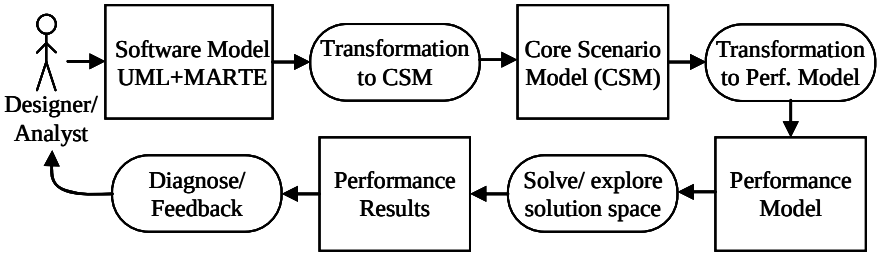


Figure 6. *PUMA transformation and performance analysis chain*

The CSM metamodel is similar to the SPT Performance Profile, describing three main types of concepts: resources, scenarios, and workloads. A scenario is a graph of steps with precedence relationships. A step may represent a basic operation or be refined as a sub-scenario. There are the following kinds of resources in CSM: a) ProcessingResource - a node in a deployment diagram; b) ComponentResource – a process or active instance related to a lifeline role in a sequence diagram or a swimlane in an activity diagram; c) LogicalResource; and d) external resource – a resource not explicitly represented in the UML model required for executing external operations that have a performance impact (for example, a disk operation).

6. Case study performance model

The performance experiments conducted with the LQN model of the TPC-W scenario from Figure 3 compares two design alternatives: one for the source model as presented in section 3.2 and the other after adding SSL secure communication between the user browser and the webserver. Both performance models include the LQN model elements generated from the *Checkout* fragment (not given in this chapter). Details on how to add security enhancements to a system model in general and to the TPC-W in special can be found in (Woodside *et al*., 2009) and (Houmb *et al*., 2010). Figure 7 shows the simplified LQN models (only the tasks and devices) for the two alternatives without and with SSL. The shaded tasks from Figure 7.b have been added to perform the SSL functionality (encryption and decryption being the most important functions) on the user and webserver side. The dotted arrows represent forwarding requests.

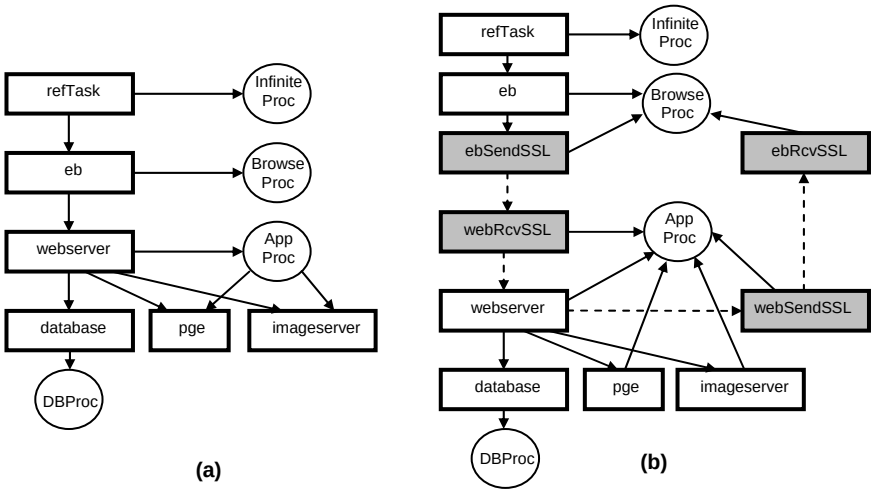


Figure 7. *LQN model for the example system: (a) without SSL; (b) with SSL.*

The results of the LQN experiments for the *GetBuyConfirmPage* scenario are shown in Figures 8. The three curves represent the response time versus the number of users for the following design/configuration alternatives:

a) The lowest curve corresponds to the initial model of the scenario without SSL. The concurrency level of the software tasks has been chosen such that the system gives the maximum performance for the given hardware configuration.

b) The highest curve corresponds to the model with SSL, for the concurrency level obtained immediately after adding SSL, without any attempt to optimize for performance. The response time has a typical non-linear shape with a “knee” around 60, after which it grows very fast due to the saturation of the system.

c) The middle curve corresponds to the SSL enhanced system with an improved software configuration. The problem before this improvement was that one of the software tasks charged with security functions on the server side becomes saturated, even though the hardware resources are not used at maximum capacity. Such a situation is known as "software bottleneck”. The solution is to increase the concurrency level, in this case to introduce more threads for the bottleneck task, in order to use the available capacity of the hardware resources. The response time improves and the new bottleneck moves to the processor running the *webserver*. The next performance solution would need to add new processing capacity at the hardware resource level.

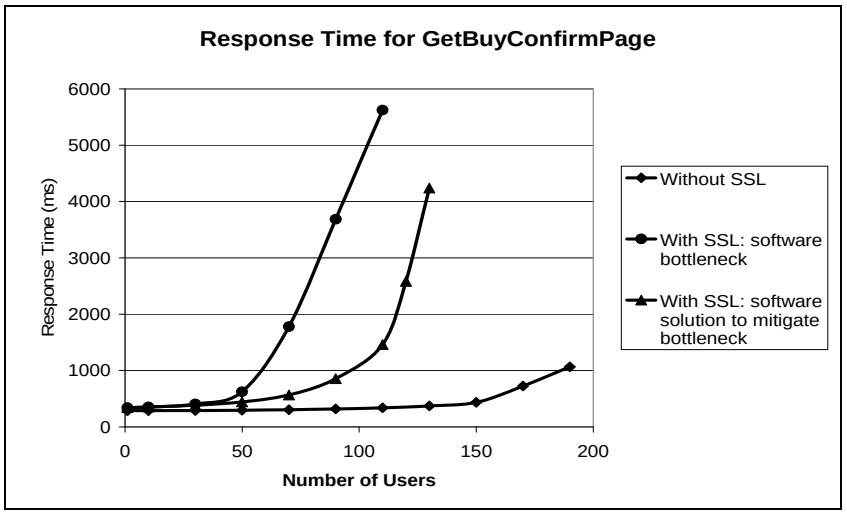


Figure 8. *Response time for GetBuyConfirmpage scenario without and with SSL*

7 Conclusions

Experience in conducting model-driven NFP analysis in the context of MDE shows that the domain is still facing a number of challenges.

Human qualifications. Software developers are not trained in all the formalisms used for the analysis of different non-functional properties (NFPs), which leads to the idea that we need to hide the analysis details from developers. However, the software models have to be annotated with extra information for each NFP and the analysis results have to be interpreted in order to improve the designs. A better balance needs to be made between what to be hidden and what to de exposed.

Abstraction level. The analysis of different NFPs may require source models at different levels of abstraction/detail. The challenge is to keep all the models consistent.

Tool interoperability. Experience shows that it is difficult to interface and to integrate seamlessly different tools, which were created at different times with different purposes and maybe running on different platforms.

Software process. Integrating the analysis of different NFP raises process issues. For each NFP it is necessary to explore the state space for different design alternatives, configurations, workload parameters in order to diagnose problems and decide on improvement solutions. The challenge is how to compare different solution alternatives that may improve some NFPs and deteriorate others, and how to decide on trade-offs.

Change propagation through the model chain. Currently, every time the software design changes, a new analysis model is derived in order to redo the analysis. The challenge is to develop incremental transformation methods for keeping different model consistent instead of starting from scratch after every model improvement.

Acknowledgements

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), through its Discovery and Strategic Projects programs.

References

Ajmone Marsan M., Balbo G., Conte G., Donatelli S. and Franceschinis G., *Modelling with Generalized Stochastic Petri Nets*, Wiley Series in Parallel Computing, John Wiley and Sons, 1995.

Balsamo S., Marzolla M., "Simulation Modeling of UML Software Architectures", *Proc. ESM'03*, Nottingham (UK), June 2003

Balsamo S., Di Marco A., Inverardi P., Simeoni M., "Model-based performance prediction in software development: a survey", *IEEE Transactions on Software Engineering*, Vol 30, No.5, pp.295-310, May 2004.

Becker S., Koziolek H., Reussner R., "Model Based Performance Prediction with the Palladio Component Model", *Proceedings of the 6th ACM Int. Workshop on Software and Performance*, Buenos Aires, Argentina, Feb. 2007, p.54-65.

Bernardi S., Donatelli S., Merseguer J., "From UML sequence diagrams and statecharts to analysable Petri net models," in *Proc. 3rd Int. Workshop on Software and Performance*, Rome, July 2002, pp. 35-45.

Cavenet C., Gilmore S., Hillston J., Kloul L., and Stevens P., "Analysing UML 2.0 activity diagrams in the software performance engineering process," in *Proc. 4th Int. Workshop on Software and Performance*, Redwood City, CA, Jan 2004, pp. 74-83.

Cortellessa V., Mirandola R., "Deriving a Queueing Network based Performance Model from UML Diagrams," in *Proc. Second Int. Workshop on Software and Performance*, Ottawa, Canada, September 17-20, 2000, pp. 58-70.

Franks G., *Performance Analysis of Distributed Server Systems*, Ph.D. Thesis, Carleton University, Systems and Computer Engineering, Report OCIEE-00-01, Jan. 2000.

Grassi V., Mirandola R., Sabetta A., "From design to analysis models: a kernel language for performance and reliability analysis of component-based systems", *Proceedings of.the 5th Int. Workshop on Software and Performance*, Palma, Spain, July 2005, p.25-36.

Hillston J., *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

Houmb S.H., Georg G., Petriu D.C., Bordbar B., Ray I., Anastasakis K., and France R.B., "Balancing Security and Performance Properties During System Architectural Design", book chapter in *Software Engineering for Secure Systems: Industrial and Research Perspectives*, H.Mouratidis (Ed)., to be published by IGI Global, 2009.

Lazowska E., Zahorjan J., Scott Graham G., Sevcik K.S., *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, 1984.

Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification (SPT),*Version 1.1, OMG document formal/05-01-02, Jan 2005.

Object Management Group, "*A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems)*", Version 1.0, OMG doc. formal/2009-11-02, Dec. 2009.

Petriu D.C., Shousha C., Jalnapurkar A., "Architecture-Based Performance Analysis Applied to a Telecommunication System", *IEEE Transactions on Software Engineering*, Vol.26, No.11, pp.1049-1065, Nov. 2000.

Petriu D.C., Shen H., "Applying the UML Performance Profile: Graph Grammar-based derivation of LQN models from UML specifications," in *Computer Performance Evaluation - Modelling Techniques and Tools*, (Tony Fields, Peter Harrison, Jeremy Bradley, Uli Harder, Eds.) LNCS Vol. 2324, pp.159-177, Springer, 2002

Petriu D.C., "Performance Analysis with the SPT Profile", *In Model-Driven Engineering for Distributed and Embedded Systems* (J. Champeau, J.P. Babau, S. Gerard, Eds.), pp. 205-224, Hermes Science Publishing Ltd., London, England, 2005.

Petriu D.C., Sabetta, A. "From UML to Performance Analysis Models by Abstraction-raising Transformation", In *From MDD Concepts to Experiments and Illustrations*, (eds. J.P. Babau J-P., Champeau J., Gerard S.), ISTE Ltd., pp.53-70, 2006.

Petriu D.B., Woodside C.M., "An intermediate metamodel with scenarios and resources for generating performance models from UML designs", *Software and Systems Modeling*, Vol.6, No.2, pp. 163-184, June 2007.

Plateau B., Atif K., "Stochastic Automata Network of Modeling Parallel Systems", *IEEE Transactions on Software Engineering*, Vol.17, No.10, p.1093-1108, 1991.

Smith C.U., *Performance Engineering of Software Systems*, Addison-Wesley Publishing Co., New York, NY, 1990.

Transaction Processing Council, "*TPC Benchmark W (Web Commerce) Specification*", Version 1.8, Feb 19, 2002.

Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar,``The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software," IEEE Trans. on Computers, Vol.44, No.1, pp. 20-34, January 1995.

Woodside C.M, Petriu D.C., Petriu D.B., Shen H, Israr T., Merseguer J., "Performance by Unified Model Analysis (PUMA)", *Proc. 5th Int. Workshop on Software and Performance WOSP'2005*, pp. 1-12, Palma, Spain, 2005.

Woodside C.M., Petriu D.C., Petriu D.B., Xu J., Israr T., Georg G., France R., Bieman J., Houmb S.H., Jürjens J., "Performance Analysis of Security Aspects by Weaving Scenarios from UML Models", *Journal of Systems and Software*, Vol.82, p.56–74, 2009.

Model-based software performance analysis introduces performance concerns in the scope of software modeling, thus allowing the developer to carry on performance analysis throughout the software lifecycle. With this book, Cortellessa, Di Marco and Inverardi provide the cross-knowledge that allows developers to tackle software performance issues from the very early phases of software development. They explain the basic concepts of performance analysis and describe the most representative methodologies used to annotate and transform software models into performance models. To this end, they go al Hence the software performance analysis cycle can be iterated to meet given performance requirements or to compare different soft-ware design alternatives. The proposed approach can be integrated with other dif-ferent methods for software performance analysis in a more general environment that can provide a set of tools for quan-titative analysis of software systems. Specically, as ob-served in (Balsamo et al., 2004a) software designers would take advantage of the combined use of different methodolo-gies to evaluate the performance of software artifacts.Â Simulation-Based Performance Modeling of UML Software Architectures. PhD Thesis TD-2004-1, Dipartimento di Informatica, Universita` Caâ€™ Foscari di Venezia, Italy, Feb. 2004. This is a list of performance analysis tools for use in software development. The following tools work based on log files that can be generated from various systems. time (Unix) - can be used to determine the run time of a program, separately counting user time vs. system time, and CPU time vs. clock time. timem (Unix) - can be used to determine the wall-clock time, CPU time, and CPU utilization similar to time (Unix) but supports numerous extensions.