Software Performance Modeling

Dorina C. Petriu, Mohammad Alhaj, and Rasha Tawhid

Carleton University, 1125 Colonel By Drive, Ottawa ON Canada, K1S 5B6
{petriu,malhaj}@sce.carleton.ca, rtawhid@connect.carleton.ca

Abstract. Ideally, a software development methodology should include both the ability to specify non-functional requirements and to analyze them starting early in the lifecycle; the goal is to verify whether the system under development would be able to meet such requirements. This chapter considers quantitative performance analysis of UML software models annotated with performance attributes according to the standard "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems" (MARTE). The chapter describes a model transformation chain named PUMA (Performance by Unified Model Analysis) that enables the integration of performance analysis in a UML-based software development process, by automating the derivation of performance models from UML+MARTE software models, and by facilitating the interoperability of UML tools and performance tools. PUMA uses an intermediate model called "Core Scenario Model" (CSM) to bridge the gap between different kinds of software models accepted as input and different kinds of performance models generated as output. Transformation principles are described for transforming two kinds of UML behaviour representation (sequence and activity diagrams) into two kinds of performance models (Layered Queueing Networks and stochastic Petri nets). Next, PUMA extensions are described for two classes of software systems: service-oriented architecture (SOA) and software product lines (SPL).

1 Introduction

The quality of many software intensive systems, ranging from real-time embedded systems to web-based applications, is determined to a large extent by their performance characteristics, such as response time and throughput. The developers of such systems should be able to assess and understand the performance effects of various design decisions starting at an early stage and continuing throughout the software life cycle. Software Performance Engineering (SPE) is an approach introduced by Smith [37], which proposes to use quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives during the development of a system. SPE promotes the integration of performance analysis into the software development process from its earliest lifecycle stages, in order to insure that the system will meet its performance objectives.

The process of building a system's performance model before the system is completely implemented and can be measured begins with identifying a small set of key performance scenarios representative of the way in which the system will be used

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 219–262, 2012.

[37]. The performance analysts must understand first the system behaviour for each scenario, following the execution path from component to component, identifying the quantitative demands for resources made by each component (such as CPU execution time and I/O operations), as well as the various reasons for queueing delays (such as competition for hardware and software resources). The scenario descriptions thus obtained can be mapped (manually or automatically) to a performance model, which can be used for By solving the model, the analyst will obtain performance results such as response times, throughput, utilization of different resources by different software components, etc. Trouble spots can be identified and alternative solutions for eliminating them can be assessed in a similar way. Many modeling formalisms have been developed over the years for software performance evaluation, such as queueing networks (QN), extended QN, Layered Queueing Networks (LQN) (a type of extended QN), stochastic Petri nets, stochastic process algebras and stochastic automata networks, as surveyed in [5][15].

Model-Driven Development (MDD) is an evolutionary step in the software field that changes the focus of software development from code to models. MDD uses *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 also the analysis of non-functional properties (NFP), by deriving analysis models for different NFPs from the software models. Ideally, analysis models should be generated automatically by model transformations from the software models used for development, and become part of the model suite which is maintained with the product. For brevity, we term the software models as *Smodels*, and the performance models as *Pmodels*.

To facilitate the generation of Pmodels, UML Smodels can be extended with standard performance annotations provided by the "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems" (MARTE) [30] defined for UML 2.x or its predecessor, the "UML Profile for Schedulability, Performance and Time" (SPT) [31] defined for UML1.x. Using UML profiles provides the additional advantage that the extended models can be processed with standard UML editors, without any need to change the tools, as profiles are standard mechanisms for extending UML models.

This chapter addresses the problem of bridging the semantic gap between different kinds of software models and performance models. We present the PUMA (Performance by Unified Model Analysis) transformation chain, whose strategy [48] "unifies" performance evaluation in the sense that it can accept as input different types of source Smodels (from which the users choose the most suitable for their project) and it generate different types of Pmodels (also according to the user's choice). To permit a user to combine arbitrary Smodel and Pmodel types according to project needs (an N-by-M problem), PUMA employs an intermediate (or pivot) language called Core Scenario Model (CSM) [34]. Based around CSM, PUMA has an open architecture summarized in Figure 1 which shows the transformers (rounded rectangles) and the flow of artifacts (rectangles) between them. It exploits several standards: UML and its model-interchange XMI standard, MARTE, performance model standards [18] [31], and the CSM metamodel [24] [25]. With suitable translators, PUMA can support other design specification language defining scenarios and resources, and other performance models.

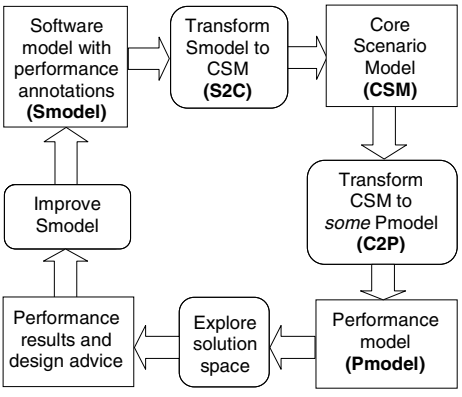

Fig. 1. PUMA transformation chain

Related Work. Many kinds of Pmodels (including queueing networks (QNs), extended QNs, stochastic Petri nets, process algebras and automata networks) can be used for performance analysis of software systems, as surveyed in [5]. The Pmodels are often constructed "by hand", based on the insight of the analysts and their interactions with the designers. To fit into MDD, the present purpose is to automate the derivation of Pmodels from the Smodels used for software development. A recent book [15] covers all the way from the basic concepts for performance analysis of software systems to describing the most representative methodologies from literature for annotating and transforming Smodels into Pmodels. For example, UML models with performance annotations (mostly SPT) containing some structural view and a certain kind of behavior diagrams have been used to generate different kinds of Pmodels: from sequence diagrams (SD) to simulation model [6], from SD and statecharts (SC) to stochastic Petri nets [11][12], from SD to QNs [16], from activity diagrams (AD) to stochastic process algebra (PEPA) [13], from SD to PEPA [44], from UML to an intermediate model called Performance Model Context (PCM) to stochastic Petri nets [20]. Many of these approaches transform from *one* kind of UML behaviour diagram plus architectural information to *one* kind of Pmodel. The difference of the PUMA strategy is that it *unifies* performance evaluation by accepting different types of source Smodels and generating multiple types of Pmodel, via the intermediate language Core Scenario Model (CSM), as described in more detail in the next sections.

Another model driven approach for development and evaluation of non-functional properties such as performance and reliability is based on the Palladio Component Model (PCM), which allows specifying component-based software architectures in a parametric way [27]. PCM captures the software architecture with respect to static structure, behaviour, deployment/allocation, resource environment/execution environment, and usage profile. Although its metamodel is completely different from UML, the Palladio Component Model has a UML-like graphical notation representing component diagrams, deployment and individual service behaviour models (similar to activity diagrams).

There are other intermediate models proposed in literature similar to PUMA's CSM, which captures only those software aspects that are relevant to performance models. An example is the pioneering "execution graph" of Smith [37], which is a kind of scenario model with performance parameters that is transformed into an extended QN model. Another intermediate language that supports performance and reliability analysis of component-based systems is KLAPER [26]. It is more oriented toward representing calls and services rather than scenarios and has a more limited view of resources (i.e., no basic distinction between hardware/software, active/passive). It has also been applied as intermediate model for transformation from different types of Smodels to different types of Pmodels.

The remaining of the paper is organized as follows: Section 2 describes how PUMA bridges the gap between Smodels and Pmodels through performance annotations and presents the source, target and intermediate models; Section 3 describes the transformations in the PUMA chain; Section 4 introduces PUMA extensions for handle service-oriented systems; Section 5 presents PUMA extensions needed to handle software product lines and Section 6 presents the conclusions and future work.

2 Source, Intermediate and Target Models

2.1 Bridging the Gap between Smodels and Pmodels

Time-related performance is a runtime property of a software system determined by how the software behaviour uses the system resources. Contention for resource creates queueing delays that directly affect the overall performance. System performance measures are closely connected with the use of the system as described by a subset of use cases with performance constraints, and more specifically by selected scenarios realizing such use cases. For instance, response time is usually defined as the end-to-end delay of a particular scenario, and throughput is the frequency of execution of a scenario or a set of related scenarios. Scenarios corresponding to online operations frequently required by customers who are waiting for the results have high priority for performance analysis, while scenarios doing housekeeping operations in the background may be less important.

The Smodel and Pmodel share similar concepts of *resources* and *scenarios*. In both, scenarios are composed of units of behaviour (called *steps*) which are using resources. Hierarchical definition of steps is possible: a step may represent an elementary operation or a whole sub-scenario. However an important difference between Smodel and Pmodel is that the first is function/data-centric, while the second is resource-centric. In other words, the Smodel scenario steps process data and implement algorithms, while the Pmodel steps care mostly about what resources are used, how and for what duration. This creates a semantic gap that needs to be bridged in the process of deriving a Pmodel from a Smodel by adding performance annotations to the latter.

Normally a Pmodel is generated from a Smodel subset containing the following:

- High-level software architecture describing the main system components instances and their interactions at a level of abstraction that captures certain characteristics relevant to performance, such as distribution, concurrency, parallelism, competition for software resources (such as software servers and critical sections), synchronization, serialization, etc.
- Allocation of high-level software components instances to hardware devices usually modeled as a deployment diagram.
- A set of key performance scenarios annotated with performance information (see section 2.3 for a concrete example).

In order to understand what kind of performance annotations need to be added to UML Smodels, we need to look at the basic concepts contained in the *performance domain model*. As already mentioned, performance is determined by how the system behaviour uses system resources. Scenarios define execution paths with externally visible end points. Performance requirements (such as response time, throughput, probability of meeting deadlines, etc.) can be placed on scenarios. In the "UML Profile for Schedulability, Performance and Time" (SPT), the performance domain model describes three main types of concepts: *resources*, *scenarios*, and *workloads* [31]. These concepts are also used in MARTE [30].

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 fixed number of users or jobs in the system).

Another source for the gap between Smodels and Pmodels is the fact that performance is a system characteristic, affected not only by the application under development represented by the Smodel, but also by the underlying platforms on top of which the application will be running (such as middleware, operating system, communication network software, hardware). There are different ways to approach this problem: one is to add the missing platform information in performance annotations, as explained in the subsections 2.3 and 2.4. Another way is to take a MDA-like approach [33], by considering that the application Smodel is a Platform-Independent Model (PIM) which can be composed with platform models defined as aspect models; the result of the composition is a Platform Specific Model (PSM). Such an approach is presented in Section 4.

2.2 MARTE Performance Annotations

In the "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems" (MARTE) [30], 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). In addition to CPU execution, a *Step* can demand the

execution of other *Scenarios*, *RequestedServices* offered by components at interfaces, and "external operations" (*ExtOp*) which are defined outside the Smodel. (ExtOp is one of the means of introducing platform resources in MARTE annotations). A new step subtype, *PassResource*, indicates the passing of a shared resource from one process to another.

In term of Resources, PAM reuses *ExecutionHost* 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.

2.3 Source Model: UML+MARTE

This section presents an example of a UML+MARTE source model for two CORBA-based client-server systems selected from a performance case study published in [1]: one is called the Handle-driven ORB (H-ORB) and the other the Forwarding ORB (F-ORB). For each case, the authors have implemented a performance prototype based on a Commercial-Off-The-Shelf (COTS) middleware product and a synthetic workload running on a network of Sun workstations using Solaris 2.6; the prototypes were measured for a range of parameters.

We used the system description from [1] to build a UML+MARTE model of each system, which represents the source model for the PUMA transformation. The results of the LQN model generated by PUMA are compared with measurement results presented in [1].The synthetic application implemented in [1] contains two distinct services A and B; the clients connect to these services through the ORB. Each client executes a cycle repeatedly, making one request to Server A and one to Server B. Two copies of A, called A1 and A2, and two copies of B, called B1 and B2, are provided. The two copies of each server enable the system to handle more load and allow the investigation of the performance effects of load balancing that is provided by many commercial ORB products. The client performs a *bind* operation before every request. The client request path varies depending on the underlying ORB architecture. In the H-ORB, the client gets the address of the server from the agent and communicates with the server directly. In the F-ORB, the agent forwards the client request to the appropriate server, which returns the results of the computations directly to the client. When a service is requested form a particular server, the server process executes a loop and consumes a pre-determined amount of CPU time. The synthetic application is used because it provides flexibility in experimentation with various levels of different workload parameters, such as the service time at each server, and the inter-node delay.

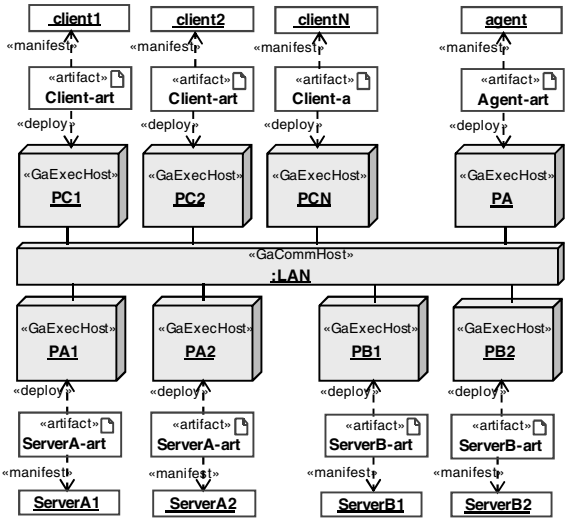

Fig. 2. The deployment of the H-ORD performance prototype

The synthetic application as considered here is characterized by the following parameters: number of clients N, service demands SA, SB representing the CPU execution time for each service, inter-node communication delay D and message length L. Since the experiments were performed on a local area network, the inter-node delay that would appear in a wide-area network was simulated by making a sender process sleeps for D units of time before sending a message. However, in the case of the H-ORB agent there was no access to the source code, so the inter-node delay for the handle returning operation was simulated by making the client sleep for D units of time before receiving the message.

Figure 2 shows the deployment diagram for the H-ORB performance prototype. The processing nodes are stereotyped as *«GaExecHost»* and the LAN communication network nodes as *«GaCommHost»*. Each client, each server and the ORB agent are allocated on their own processor.

Figure 3 represents the client request scenario in the form of a sequence diagram (SD), while Figure 4 represents the same scenario as an activity diagram (AD). Both the SD and the AD are stereotyped with *«GaAnalysisContext»* that indicate that the respective scenarios are to be considered for performance analysis. Each lifeline role stereotyped by *«PaRunTInstance»* is related to a runtime concurrent component instance, which is in turn allocated on a processor in the deployment diagram. The first step of the scenario has a workload stereotype *«GWorkloadEventt»* with an attribute pattern indicating that the scenario is used under a closed workload with a population of $N. ($N indicates a MARTE variable, to be substituted by a concrete value when the performance model is actually solved. By convention, the name of all MARTE variables in this work begin with "$" to distinguish them from other names). A *«PaStep»* stereotype is applied to each of the steps corresponding to the following messages: Get-Handle(), A1Work(), A2Work(), B1Work() and B2Work(). All scenario steps are characterized by a certain *hostDemand*, which represents the CPU execution time.

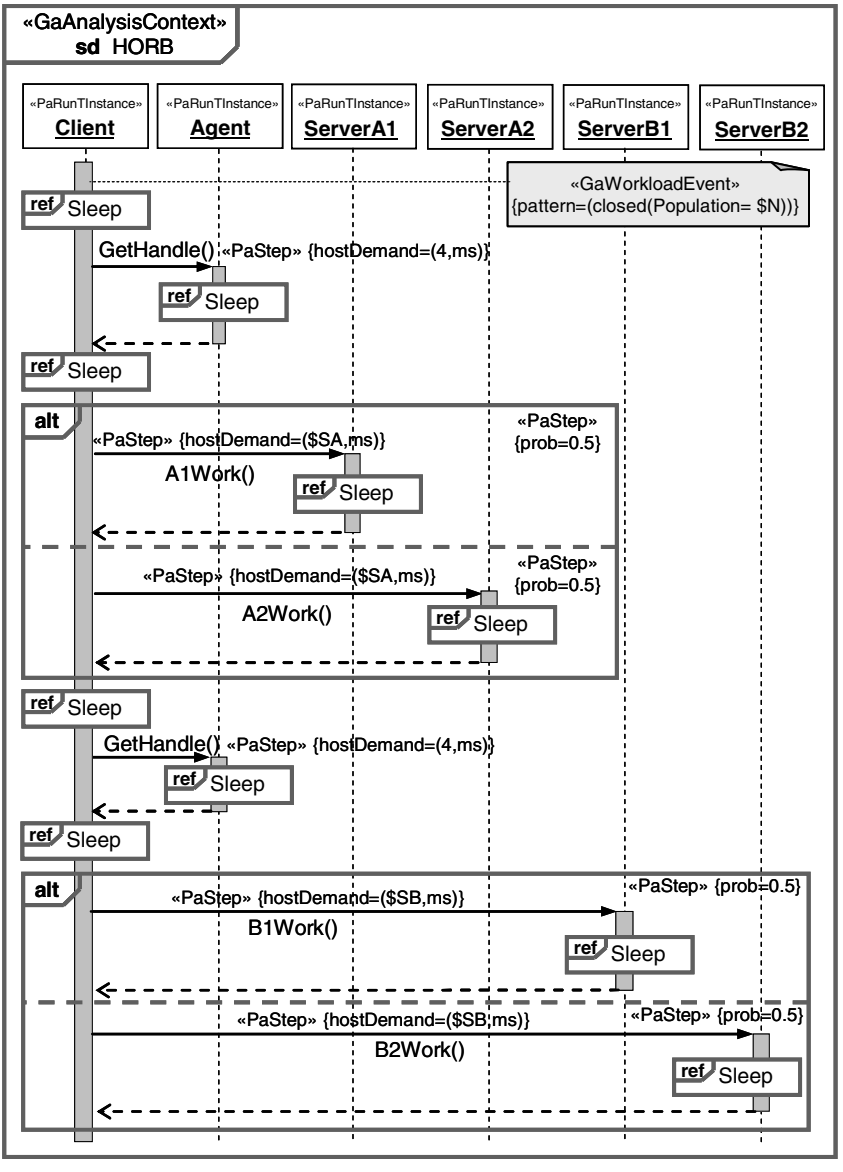

Fig. 3. Client request scenario for H-ORB as a sequence diagram

In the SD from Figure 3, the choice of which server instance to call (A1 or A2; B1 or B2) is modeled as two *alt* combined fragments respectively, with two operands each. An operand itself is a *«PaStep»* with the attribute *prob* indicating the probability of being chosen. The call to the Sleep() function in the SD is modeled by an interaction occurrence *ref* making reference to another SD not shown here, which

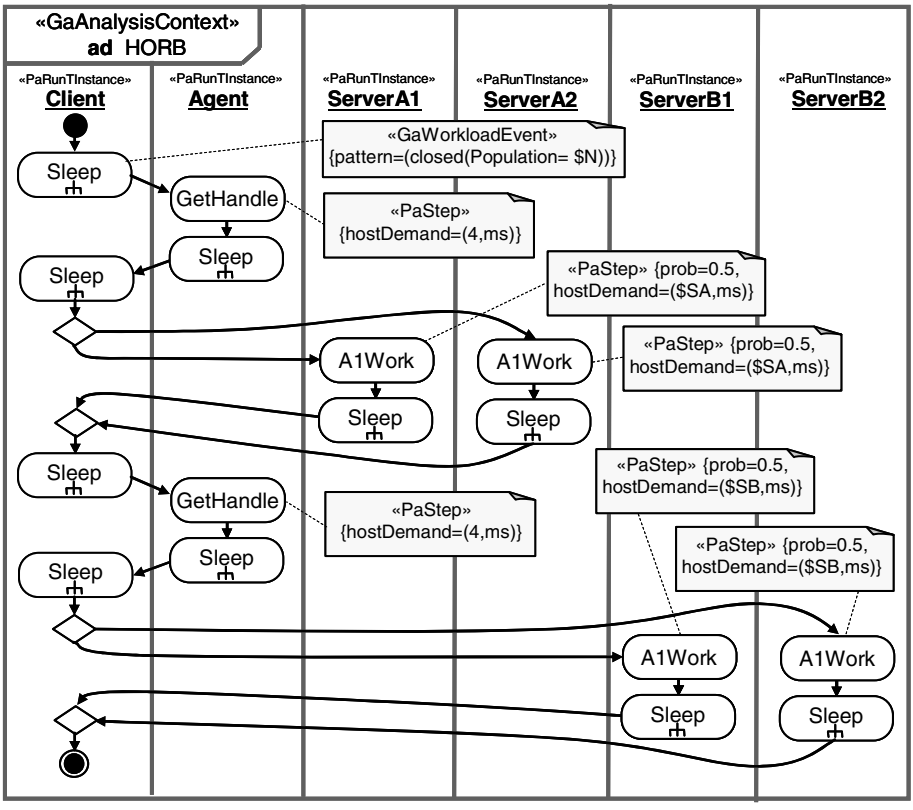

Fig. 4. Client request scenario for H-ORB as an activity diagram

contains a call to a dummy server Sleep that delays the caller by a required time without consuming the CPU time of the caller.

In the activity diagram, every active concurrent instance is represented by its own partition (a.k.a. swimlane) and is stereotyped by *«PaRunTInstance»*. The scenario steps modeled as activities are stereotyped as *«PaStep»* with the same *hostDemand* and *prob* attributes as in SD. An AD arc crossing the boundary between partitions represents a message sent from one active instance to another. Sleep is a structured activity, which contains inside the details of a call to a dummy instance that delays the caller without consuming its CPU time. Eventually, in the performance model Sleep will be represented as a dummy server that performs the same roll.

The scenario for the F-ORB case study is not presented here, but it is fairly similar with the H-ORB. In the F-ORB architecture the client sends the entire service request to the agent that locates the appropriate server and forwards the request to it. The server performs the desired service and sends a response back to the client.

After describing the UML source model extended with MARTE annotations, we will present the target performance model and then the intermediate model used in the PUMA transformation.

2.4 Target Performance Model: LQN

Many performance modeling formalisms have been developed over time, such as queueing networks (QN), extended QN, Layered Queueing Networks (LQN), stochastic Petri nets, stochastic process algebras and stochastic automata networks. Although PUMA can incorporate model transformations from CSM to nay performance modeling formalisms, in the paper we will consider one target performance model, the Layered Queueing Network (LQN) [46][36].

LQN was developed as an extension of the well-known Queueing Network model; the main difference is that LQN can easily represent nested services: a server may become in turn a client to other servers from which it requires nested services, while serving its own clients. The LQN toolset presented in [22][23] includes both simulation and analytical solvers.

A slightly simplified LQN metamodel is presented in Figure 5. Examples of LQN models are presented in Figures 5 and 18.

A LQN model is an acyclic graph, with nodes representing software entities and hardware devices (both known as *tasks*), and arcs denoting service requests. The software entities are drawn as rectangles with thick lines, and the hardware devices as ellipses. 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 (such as processors, I/O devices, communication network, etc.) A software or hardware server node can be either a single-server or a multi-server.

Each kind of service offered by a LQN task is modeled as an *entry*, drawn as a rectangle with thin lines attached to the task or other entries of the same task. Every entry has its own execution times and demands for other services (given as model parameters). Each software task is running on a processor shown as an ellipse. The communication network, disk devices and other I/O devices are also shown as ellipses. The word “layered” in the LQN name does not imply a strict layering of tasks (for example, tasks in a layer may call each other or skip over layers). The arcs with a filled arrow represent synchronous requests, where the sender is blocked until it receives a reply from the provider of service. It is possible to have also asynchronous request messages (shown as a stick arrow), where the sender does not block after sending a request and the server does reply back. Another communication style called *forwarding* (shown with a dotted line), allows for a client request to be processed by a chain of servers instead of a single server. The first server in the chain will forward the request to the second and become free; the second to the third, etc., and the last server in the chain will reply to the client. Although not explicitly illustrated in the LQN notation, every server, be it software or hardware, has an implicit message queue, where incoming requests are waiting their turn to be served. Servers with more then one entry have a single input queue where requests for different entries wait together.

A server entry may be decomposed in two or more sequential phases of service. Phase 1 is the portion of service during which the client is blocked waiting for a reply

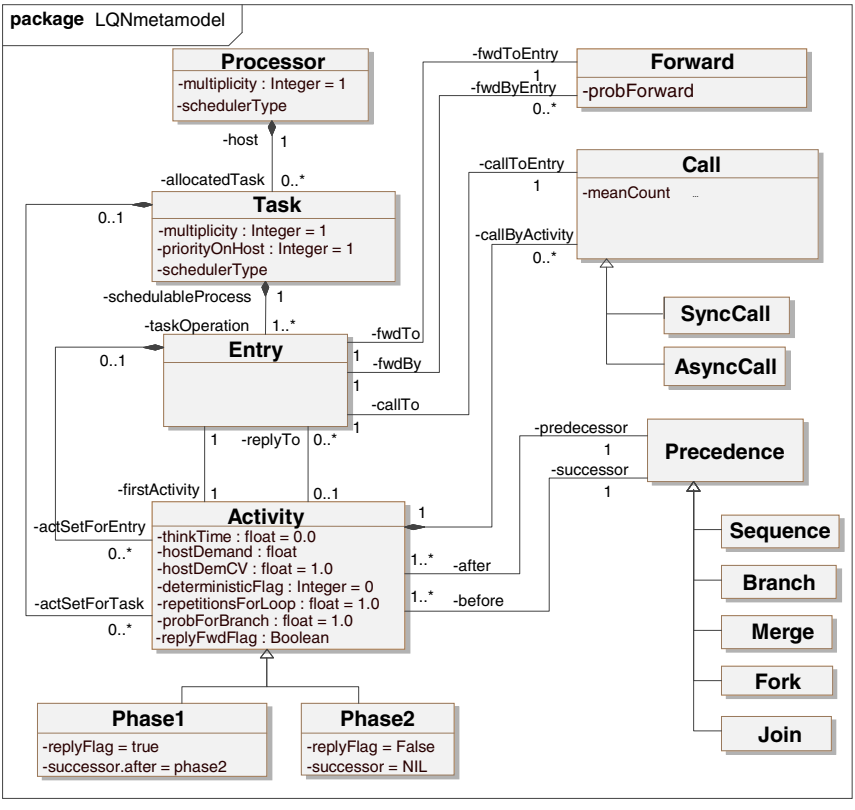

Fig. 5. LQN metamodel

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 [23] allows for an entry to be further decomposed into activities if more details are required to describe its execution (see Figure Y). The activities are connected together to form a directed graph that may branch into parallel threads of control, or may choose randomly between different branches. Just like phases, activities have execution time demands, and can make service requests to other tasks.

The parameters of a LQN model are as follows:

- customer (client) classes and their associated populations or arrival rates;
- for each phase (activity) of a software task entry: average execution time;
- for each phase (activity) making a request to a device: average service time at the device, and average number of visits;
- for each phase (activity) making a request to another task entry: average number of visits
- for each request arc: average communication delay;
- for each software and hardware server: scheduling discipline.

2.5 Intermediate Model: CSM

The Core Scenario Model [34] represents 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. The CSM metamodel is similar to the SPT Performance Profile, describing three main types of concepts: *resources*, *scenarios*, and *workloads*. Each *Scenario* is a directed graph with *Steps* as nodes, and explicit *PathConnectors* which define *Sequence, Branch, Merge, Fork* and *Join*. A *Step* is owned by a *Component*, which may be a *ProcessResource*, and which in turn is associated to a *HostResource* (processor). *Logical resources* are acquired and released along the path by special subtypes of *Step* called *ResourceAcquire* and *ResourceRelease*. *External Resource* represents a resource not explicitly represented in the UML model required for executing external operations that have a performance impact (for example, a disk operation). The CSM metamodel is described in more detail in [34].

3 PUMA Transformation Chain

In this section we present the principles of the transformation used in PUMA: a) from source Smodel in UML extended with MARTE to the intermediate CSM; and b) from CSM to LQN Pmodel. The section also shows a few performance results obtained with the LQN model generated from the CORBA source model introduced in Section 2.3 and compares them with measurements.

3.1 Transformation from UML+MARTE to CSM

The general strategy is to identify the scenarios and structural diagrams to be considered by looking for MARTE stereotypes and then to generate structural CSM elements (*Resources* and *Components*) from the structure diagram (e.g., deployment),

Table 1. Mapping between MARTE stereotypes and CSM Elements

MARTE	CSM
«GaWorkloadEvent»	Closed/OpenWorkload
«GaScenario»	Scenario
«PaStep»	Step
«PaCommStep»	Step (for the message)
«GaResAcq»	ResourceAcquire
«GaResRel»	ResourceRelease
«PaResPass»	ResourcePass
«GaExecHost»	ProcessingResource
«PaCommHost»	ProcessingResource
«PaRunTInstance»	Component
«PaLogicalResource»	LogicalResource

and behavioural elements (*Scenarios*, *Steps* and *PathConnectors*) from the behaviour diagrams. The mapping between MARTE stereoptypes and CSM elements is presented in Table 1.

The transformation algorithm begins with generating the structural elements first. A UML Node from a deployment diagram stereotyped *«GaExecHost»* or *«PaCommHost»* is converted into a CSM *ProcessingResource.* A UML run-time component manifested by an artifact, which is in turn deployed on a node is converted into a CSM *Component*.

Scenarios Described by Sequence Diagrams. The transformation continues with the scenarios described by sequence diagrams stereotyped with *«GaAnalysisContext»*. For each scenario, a CSM *Start PathConnection* is generated first, and the workload information is attached to it. Each *Lifeline* from a sequence diagram describes the behaviour of a UML instance (be it active or passive) and corresponds in turn to a CSM *Component*. The *Lifelines* stereotyped as *«PaRunTInstance»* corresponds to an active runtime instance. We assume that the artifacts for all active UML instances are shown on the deployment diagram, so their corresponding CSM *Components* were already generated. However, it is possible that the sequence diagram contains lifelines for passive objects not shown in the deployment diagram. In such a case, the corresponding CSM *Passive Component* is generated, and its host is inferred to be the same as that of the active component in whose context it executes.

The translation follows the message flow of the scenario, generating the corresponding *Steps* and *PathConnections*. A simple *Step* corresponds to a UML *Execution Occurrence*, which is the execution of an operation as an effect of receiving a message. Complex CSM Steps with a nested scenario correspond to operand regions of UML *Combined Fragments* and *Interaction Occurrences*. A synchronous message will generate a CSM *Sequence PathConnection* between the step sending the message and the step executed as an effect. An asynchronous message spawns a parallel thread, and thus will generate a *Fork PathConnection* with two outgoing paths: one follows the sender's activity, and the other follows the path of the message. The two paths may rejoin later through a *Join PathConnection*. Fork/join of parallel paths may be also generated by a *par Combined Fragment*. Conditional execution of alternate paths is generated by *alt* and *opt Combined Fragments*.

Scenarios Described by Activity Diagrams. We consider all the scenarios described by activity diagrams stereotyped as *«GaAnalysisContext»*. For each scenario, the transformation starts with the *Initial ControNode,* which is converted into a CSM *Start PathConnection* and a *Resource Acquire* step for acquiring the component for the respective swimlane. Also, the scenario workload information described by a *«GaWorkloadEvent»* stereotype is used to generate a CSM *Workload* element attached to the *Start PathConnection*. (Note that in MARTE, the scenario workload information is associated by convention with the first step of a scenario, not with its *Initial ControlNode,* which cannot be stereotyped as *Step*). The translation follows the sequence of the scenario from start to finish, identifying the *Steps* and *PathConnections* (sequence, branch/merge, fork/join) from the context of the diagram. Each UML *ActivityNode* that represents a simple activity is converted into a CSM *Step*, one that represents an activity further refined by another diagram generates a CSM *Step* with a nested *Scenario*.

As mentioned before, we assume that each partition (a.k.a. swimlane) is associated with a *Component* through the *«PaRunTInstance»* stereotype. A special treatment is given to *ActivityEdges* that cross the partition boundary (named here cross-transition). A cross-transition represents a message (signal) between the corresponding components that implies releasing the sender (which is a *Componen*t, but also a *Resource*) and acquiring the receiver. Therefore, a cross-transition generates in CSM a *ResourceRelease* step, a *Sequence PathConnection* and *ResourceAcquire* step.

Figure 6 shows the CSM generated for the H-ORD source model from section 2.3.

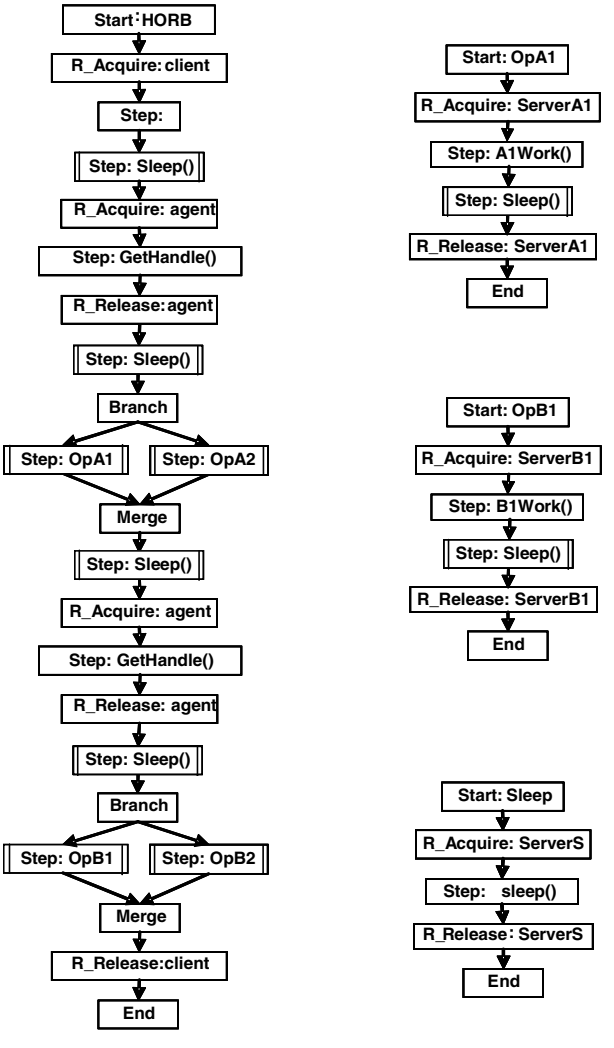

Fig. 6. CSM for the H-ORB system from Figures 2 and 3

The main CSM model represents the main flow of steps from the scenario represented in Figure 3 as SD and in Figure 4 as AD. Composite steps were generated for every *Interaction Occurrence* invoking Sleep() and for each operand of the *alt CombineFragments* making a choice of a server. The Composite steps are refined by CSM sub-scenarios on the right of the figure. (The fragment for the operand invoking A2 is not shown, being similar with the one invoking A1; the same is true for operand invoking B2, which is similar to B1).

3.2 Transformation from CSM to LQN

The first stage of the transformation algorithm parses the resources in the CSM and generates a LQN *Processor* for each CSM *ProcessingResource* and an LQN *Task* for each CSM *Component*. The second stage traverses the CSM to determine the branching structure and the sequencing of *Steps* within branches, and to discover the calling interactions between *Components.*

The traversal creates a new LQN *Entry* whenever a task receives a call. The entry internals are described by LQN *Activities* that represent the sequence of *Steps* for the call, using a notation like CSM itself. Another possibility is to generate LQN *Phases* when there is only a sequence of Steps (without branching or forking). The traversal generates an LQN *Activity* for each CSM *Step* it encounters and it generates LQN *Branch, Merge, Fork, Join* and *Sequence* connectors corresponding to the same *PathConnectors* in the CSM. Whenever an interaction between two CSM *Components* is detected, an *Activity* is created in the *Task* corresponding to the requesting *Component* with a *Call* to the new LQN *Entry* which is created in the *Task* corresponding to the called *Component*. This *Entry* serves the request and its workload is defined by the ensuing *Activities* generated from the *Steps* encountered in the new *Component*.

The type of call (synchronous or asynchronous) is detected by its context in the CSM. More exactly, a message back to a *Component* that previously sent a request is considered to be a reply to a synchronous call. Any messages that do not have matching replies when the end of the scenario is reached are considered to be asynchronous calls. During the traversal of the CSM, the algorithm creates a stack of unresolved call messages and removes them as the matching reply messages are detected (other interaction patterns can also be identified). At *Branch* and *Fork* points, the stack of unresolved messages is duplicated for each outgoing alternate or parallel subpath so that each ensuing subpath maintains its own message history. All of the duplicate call stacks except one are discarded at *Merge* and *Join* points after every incoming alternate or parallel branch has been traversed. The ordering of the messages is a direct result of the traversal of the CSM scenarios and is a partial order for the particular path being traversed. Parallel or alternate branches each have a partial order of the messages along their own subpaths, but no global ordering is implied.

A CSM *ClosedWorkload* is transformed into parameters for a load-generating *Reference Task*, and a CSM *OpenWorkload* into an open stream of requests made to the first entry. An *External Operation* by a CSM Step is represented by an activity which makes a call to a submodel that has to be provided by the analyst.

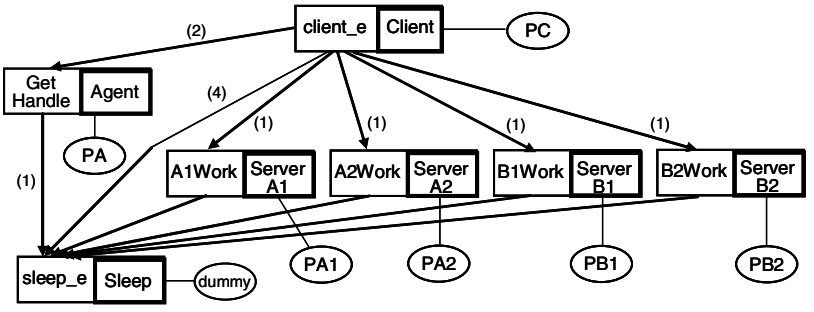

Fig. 7. LQN model for the H-ORB system

Figure 7 shows the LQN model generated from the CSM for the H-ORB given in Figure 6. As mentioned before, the Sleep task running on a dummy server implements the sleep function. All requests are synchronous calls in this example. The numbers in parentheses on the arcs represent the average number of calls. The service times (Not shown in the figure) are represented by the variables $A, $B, $D which are assigned concrete values doing the experiments.

The LQN model thus generated has been validated against measurements of the H-ORB and F-ORB performance prototypes that have been published in [1]. As it can be seen in Figure 8, the accuracy of the analytic model is fairly reasonable.

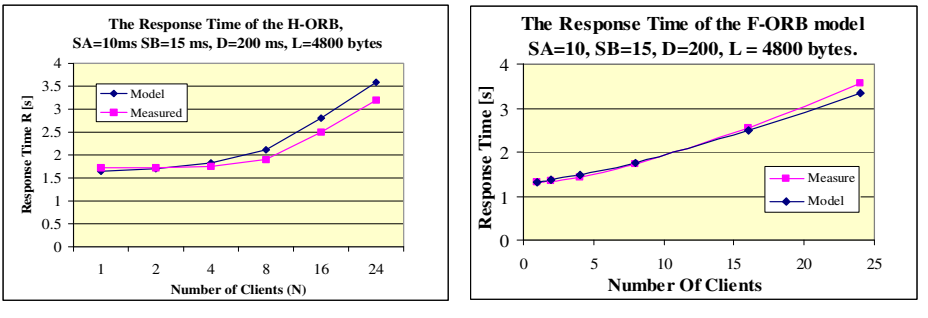

Fig. 8. Validation of the LQN results against measurements

In this section we have presented the PUMA transformations from a Smodel to the corresponding intermediate model to the Pmodel. According to the PUMA architecture from Figure 1, once the Pmodel has been generated, the next step is to use it for experiments that are exploring the parameter space in order to evaluate design changes such as execution in parallel, replication, modified concurrency, and reduced demands and delays. The Pmodel results evaluate the potential of these changes, which can then be mapped to possible software solutions [49].

In the next two sections we will present extensions to the PUMA transformation to specialize it for Service-Oriented Architecture and for Software Product lines.

4 Extension of PUMA to Service-Oriented Architecture(SOA)

SOA is a paradigm for developing and deploying business applications as a set of reusable services [19]. SOA is used for enterprise systems, web-based applications, multimedia, healthcare, etc. Model Driven SOA (MDSOA) is an emerging approach for developing service-oriented applications developing models at multiple levels of abstraction, which can be used eventually to generate code. MDSOA is also used to verify the non-functional properties (NFP) by transforming the software models to different NFP analysis models (including performance). In order to improve modeling SOA systems, OMG has introduced a new profile called Service Oriented Architecture Modeling Language (SoaML) [32], which extends UML with the ability to model the service structure and dependencies, to specify service capabilities and classification, and to define service consumers and providers.

The emergence of MDD in general and of MDSOA in particular has attracted a lot of interest in the research community in using software models to evaluate the non functional properties of service-based systems. A model transformation framework is proposed in [45] to automatically include the architectural impact and the performance overhead of the middleware layer in distributed systems. This allows one to model the application independent of the middleware and then obtain a platform specific model by composition. Another model-driven approach for development and evaluation of non-functional properties such as performance and reliability is based on the Palladio Component Model (PCM), which allows specifying component-based software architectures in a parametric way [27]. A parametric performance completion for message-oriented middleware proposed for PCM in [27] allows for the composition of platform components with application components. Other research on building performance models for web services takes a two layered user/provider approach in [18] and [28]: the user is a represented by a set of workflows and the provider by a set of services deployed on a physical system. Performance information about service capabilities and invocation mechanisms is given by the means of P-WSDL (Performance-enabled WSDL) in [18], where a LQN model is generated for analyzing the system performance. In [28] the queueing network formalism is used to derive performance bounds.

4.1 PUMA4SOA Transformation Chain

Performance by Unified Model Analysis for SOA (PUMA4SOA) is a modeling approach proposed first in [2] which extends the PUMA transformation, specializing it for service-based systems. The difference between the original PUMA and the extended one for SOA stems from: a) the kind of design models accepted as input, and b) the separation between Platform Independent Model (PIM) and Platform Specific Model (PSM) of the application and the use of platform models. Figure 4.1 illustrates the steps of PUMA4SOA; the top leftmost represents the main difference from PUMA (whose steps are shown in Figure 1). There are three input models to PUMA4SOA: a) application PIM, b) deployment model which describes the allocation of the artifacts to a deployment target, and c) platform aspect models.

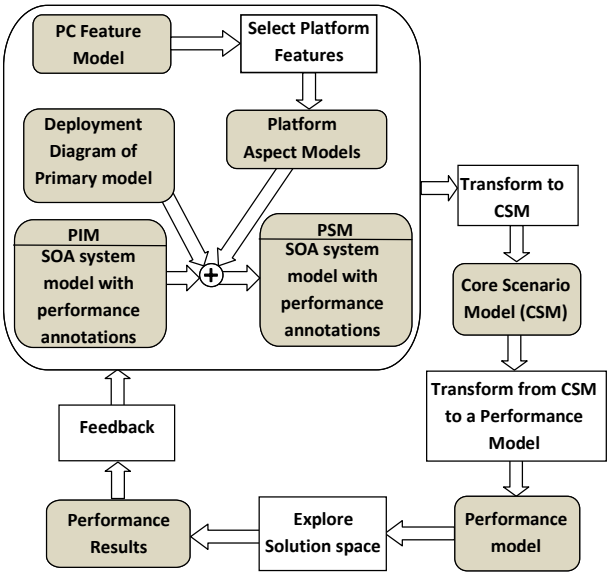

Fig. 9. The Steps of PUMA4SOA

The platform independent model of the application contains a UML software model with three levels of abstractions. The UML model is annotated with performance information using the standard UML profile MARTE. Each level represents a part of the system details that will be used together with the other parts to build the performance model. The three abstraction levels are as follows:

a) *Workflow Model* which represents a set of business processes. Each workflow contains a sequence of activities and actions controlled by conditions, iterations, and concurrency.
b) *Service Architecture Model* which describes the service capabilities arranged in a hierarchy showing anticipated usage dependencies. It also depicts the level of service granularity, which has a substantial effect on the system performance. Invoking services in heterogeneous and distributed environment produces message overheads due to marshalling/unmarshalling of the message data at the service platform. A coarser service granularity reduces the number of service invocations, which improves the performance, but produces unnecessary coupling between the components of the SOA system. However, a finer service granularity increases the number of service invocations, which reduces the performance but produces a loosely coupled system. Service Architecture Modeling helps the modeler to manage the tradeoff between the service granularity and performance.

c) *Service Behavior Model* which refines the workflow behavior, giving more details about the services invoked. Each workflow activity may be refined by a sequence diagram which represents its detailed behavior, including the invocations of the other services and the interaction between participants.

PUMA4SOA also defines two models: a) Performance Completion Feature model (PC feature model), and b) Platform aspect models. The concept of "performance completions" was introduced by Woodside et al. [47] to close the gap between abstract design models and external platform factors. The PC feature model, introduced in the work of the Palladio group (see [27]) and also used in [41], defines the variability in platform choices, execution environments, types of platform realizations, and other external factors that have an impact on the system's performance. Since the regular notation for feature diagrams is not part of UML, we use a UML class diagram extended with stereotypes to represent the PC feature model, where each feature is represented as a class element. Four relationships between a feature and its sub features are defined: Mandatory, Optional, Or, and Alternative. Each feature in the feature model represents a platform aspect. A platform aspect model describes the structure and the behavior of the service platform in a generic format. The PC feature model allows the modeler to select between different platform aspect models that are most appropriate for the application of interest.

The selected platform aspect models composed with the PIM generate the PSM. The Aspect Oriented Modeling (AOM) approach is used to generate the platform specific model (PSM) by weaving the selected platform aspect model behaviors into different locations of the platform independent model (PIM). The AOM approach requires two types of models: a) the primary model which describes the core design decisions, and b) a set of aspect models, each describing a concern that crosscuts the primary model [21]. PUMA4SOA considers the application PIM as the primary model. An aspect model can be seen as a template or pattern, independent of any primary model it may be composed with. For each composition with the primary model, the template is instantiated and its formal parameters are bound to concrete values using binding rules, to give a context-specific aspect model. The composed model is generated by weaving the context-specific aspect models into the primary model at different locations. In the next section, we will describe the PUMA4SOA approach with an example from the healthcare domain.

4.2 Platform Independent Model: Case Study

The platform independent model is illustrated with a healthcare case study, the Eligibility Referral System, which is introduced in [3]. A UML activity diagram is used to model the workflow in Figure 10. It is the top level model that describes the process of transferring a patient from one hospital to another. Three organizations are involved, the transferring and receiving hospitals and the insurance company. The workflow begins with the transferring hospital filling and processing the initial forms needed to transfer the patient. The next process is getting the physician and the payment approvals. The transferring hospital is then sending the forms and waits for

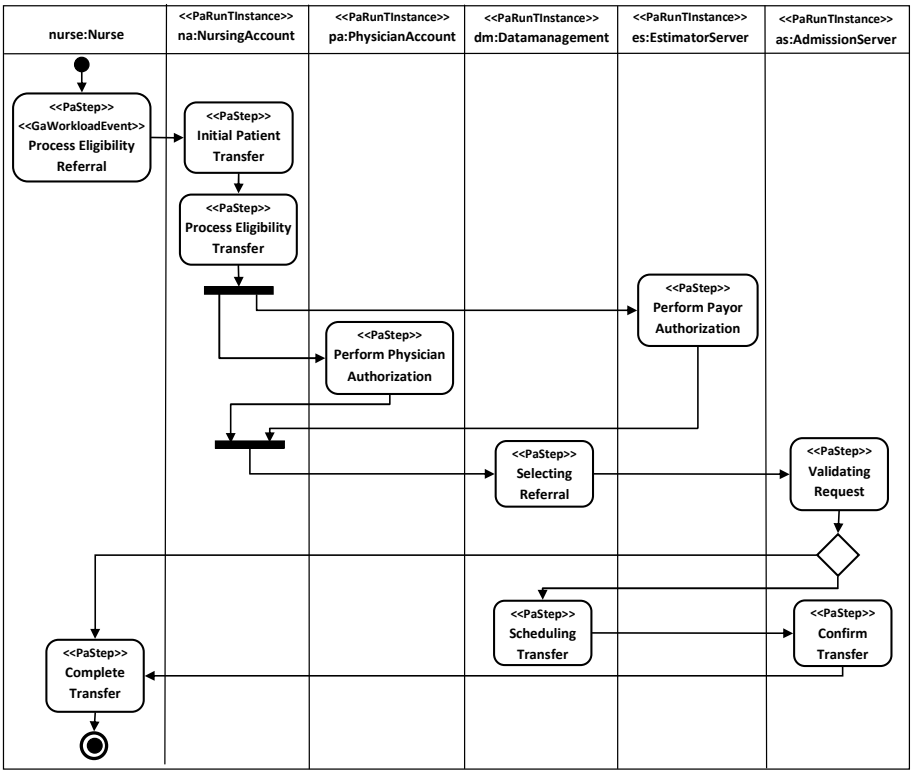

Fig. 10. Workflow model represented by an activity diagram

an acknowledgement from the receiving hospital. Finally, the transferring hospital schedules the transferring date and updates the transferring process. The workload of the system is described by the *«GaWorkloadEvent»* stereotype, which can be a closed arrival pattern defining a fixed populations of users or an open arrival pattern which defining a stream of requests that arrive at a given rate. A swimlane is stereotyped as *«PaRunTInstance»* to indicate that the activities are executed by a concurrent participant. This stereotype has a *poolsize* attribute to define the number of concurrent threads. An activity is stereotyped as *«PaStep»* to indicate a scenario step. It has a *hostDemand* attribute for the required execution time, a *prob* for its probability, and a *rep* for the number of repetitions. The communication between participants is described by *«PaCommStep»* to indicate the conveyance of a message. It has a *msgSize* attribute to indicate the amount of transmitted data.

The Service Architecture model, illustrated in Figure 11, is using a new OMG profile called the Service Oriented Architecture Modeling Language (SoaML) [32]. SoaML extends UML with the ability to define the service structure and dependencies to specify service capabilities, and to define service consumers and providers. The Eligibility Referral System defines five components stereotyped as *«participants»:*

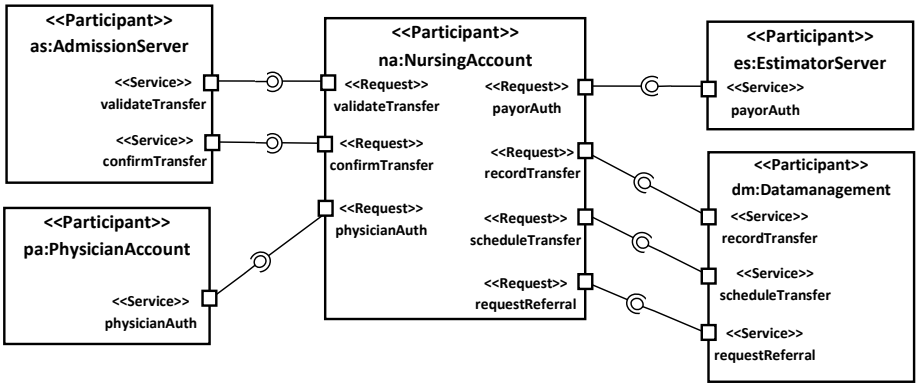

Fig. 11. Service Architecture Model

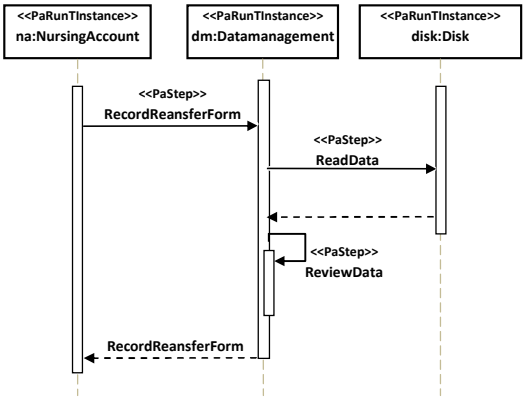

Fig. 12. Service behavior model for Initial Patient Transfer service

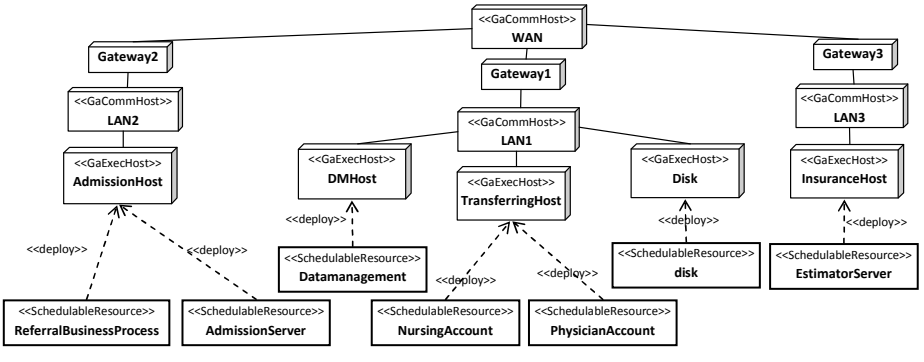

Fig. 13. Deployment of the Eligibility Referral System

NursingAccount, PhysicianAccount, AdmissionServer, EstimatorServer, and Data management. The model also presents different service contracts; each one of them defines service consumers (their ports are stereotyped with *«Request»*), and service providers (their ports are stereotyped with *«Service»*). UML sequence diagrams are used to model the behavior of each activity defined in the workflow model. Figure 12 shows an example of the service behavior model of "Initial Patient Transfer" activity. In this interaction, the nurse generates the patient transfer form by retrieving patient's data from the Database, and then reads and reviews it before sending it back to the form. Lifelines are stereotyped with *«PaRunTInstance»* to indicate a concurrent process, and messages are stereotyped with *«PaStep»* to indicate an action.

The UML Deployment diagram in Figure 12 shows the allocation of software to hardware. Physical communication nodes, such as WAN, LAN1, LAN2, and LAN3, are stereotyped with *«GaCommHost»* to indicate a physical communication link. Processors, such as *AdmissionHost, TransferringHost, InsuranceHost, DMHost,* and *Disk*, are stereotyped with *«GaExecHost»* to indicate the processor host. Artifacts, such as *NursingAccount, PhysicainAccount, AdmissionServer, EstimatorServer, Datamanagement*, and *disk*, are stereotyped with *«SchedulableResource»* to indicate a concurrent resource. The *ReferralBusinessProcess* component represents the execution engine which runs the business process of the system.

4.3 PC Feature Model

The PC feature model describes the variability in service platform which may affect the system's performance. Figure 13 describes the features which may affect the performance of our example, the Eligibility Referral System. There are three mandatory feature groups which are required by any service platform: the operation, message protocol and realization. There are also two optional feature groups: communication and data compression. The relationship between the feature groups and their sub-features are alternative with exactly-one-of feature selected. Although the dependencies between the sub-features are not shown in the model, some features, such as the operation feature, message protocol feature and realization feature, are dependent. As an example selecting one of the operation sub-features, such as invocation, requires selecting one of the message protocol (Http or SOAP) and the realization (WebService, REST, etc.)

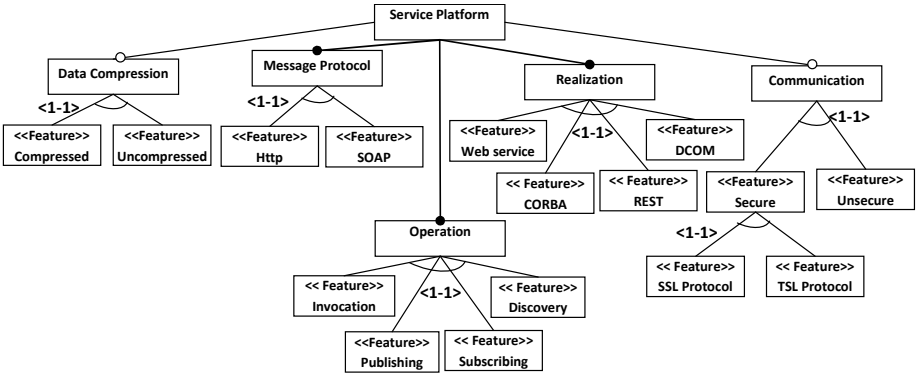

Fig. 14. Platform Completion (PC) Feature Model

4.4 Aspect Platform Model for Service Invocation

The aspect platform models define the middleware structure and behavior of the selected aspects from the PC feature model. In our example, we selected a service invocation aspect realized as a webservice with the message protocol SOAP. Figure 15 describes the generic deployment including the hosts and artifacts involved in the service invocation aspect model. As a naming convention the vertical bar 'l' indicate a generic role name as in [21]. Two hosts are involved in the service invocation operation, the |*Client* which consumes the service, and the |*Provider* which provides it.

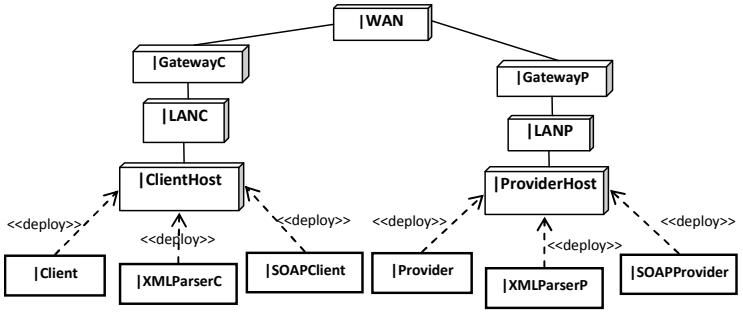

Fig. 15. Generic Invocation aspect: deployment view

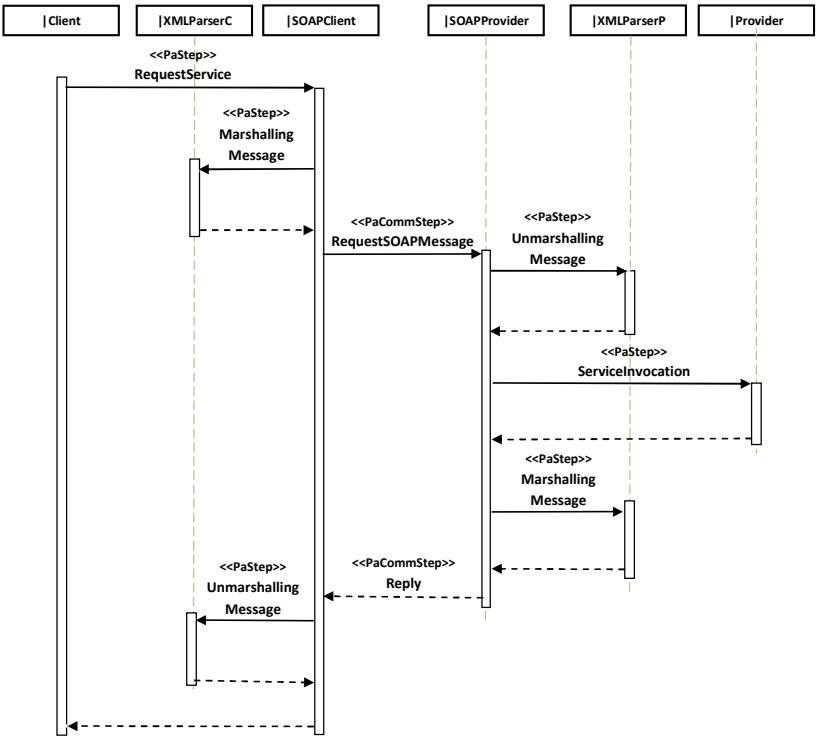

Fig. 16. Generic Aspect model for Service Invocation

The middleware on both sides contains an |*XMLParser* to marshal/unmarshal the message, and a |*SOAP* stub for message communication. Figure 16 describes the generic service request invocation and response behavior. A request message call is sent from a |*Client* to a |*Provider*. This message call differs from the regular operation call due to the heterogeneous environment it operates in, which require the message to be parsed in acceptable format at both |*Client*, and |*Provider* sides, before it is being sent or received.

In the AOM approach, the generic aspect model for the service invocation, illustrated in Figure 16, may be inserted in the PIM multiple times wherever there is a service invocation. For each insertion, the generic aspect model is instantiated and its formal parameters are bound to concrete values using binding rules to produce a context specific aspect model. Each context specific aspect model is then composed into the primary model, which in our case is the PIM. More details about AOM approach can be found in [2][48].

In PUMA4SOA the aspect composition can be performed at three modeling levels: the UML level, the CSM level or the LQN level. The complexity of the aspect composition may determine where to perform it. At the UML level, the aspect composition is more complex because the performance characteristics of the system are scattered between different views and diagrams, which may require many models to be used as input to the composition. On the contrary, performing aspect composition at the CSM or LQN level is simpler because only one view is used for modeling the system. In our example, we performed aspect composition at the CSM level which is discussed in the next section (see also [48]).

4.5 From Annotated UML to CSM

In PUMA4SOA, generating the Platform specific model can be delayed to the CSM level. The UML PIM and platform aspect models are first transformed to CSM models. The CSM PSM is then generated by composing the CSM platform aspect models with the CSM PIM. The generated CSM model is separated into two layers, the business layer representing the workflow model, and the component layer representing the service behavior model. The workflow model is transformed into the top level scenario model. The composite activities in the workflow are refined using multiple service behavior models, which are transformed into multiple sub-scenarios within the top level scenario. The CSM on the left in Figure 17 is the top level scenario representing the workflow model. It has a *Start*, and *End* elements for beginning and finishing the scenario. The *ResourseAcquire* and *ResourseRelease* indicate the usage of the resources. A *Step* element describes an operation or action. An atomic step is drawn as a box with a single line and a composite step as a box with double lines on the sides. The scenario on the right of Figure 17 illustrates the composed model which describes the PSM of the sub-scenario *InitialPatientTransfer*. The grayed parts originated from the context-specific aspect model which was composed with the PIM. Whenever a consumer requests a service in the workflow model, the generic service invocation aspect is instantiated using binding roles to generate the context specific service invocation aspect, which is then composed with the PIM to produce the PSM. Figure 10 shows seven service requests, which means that seven invocation aspect instances are composed within top level scenario.

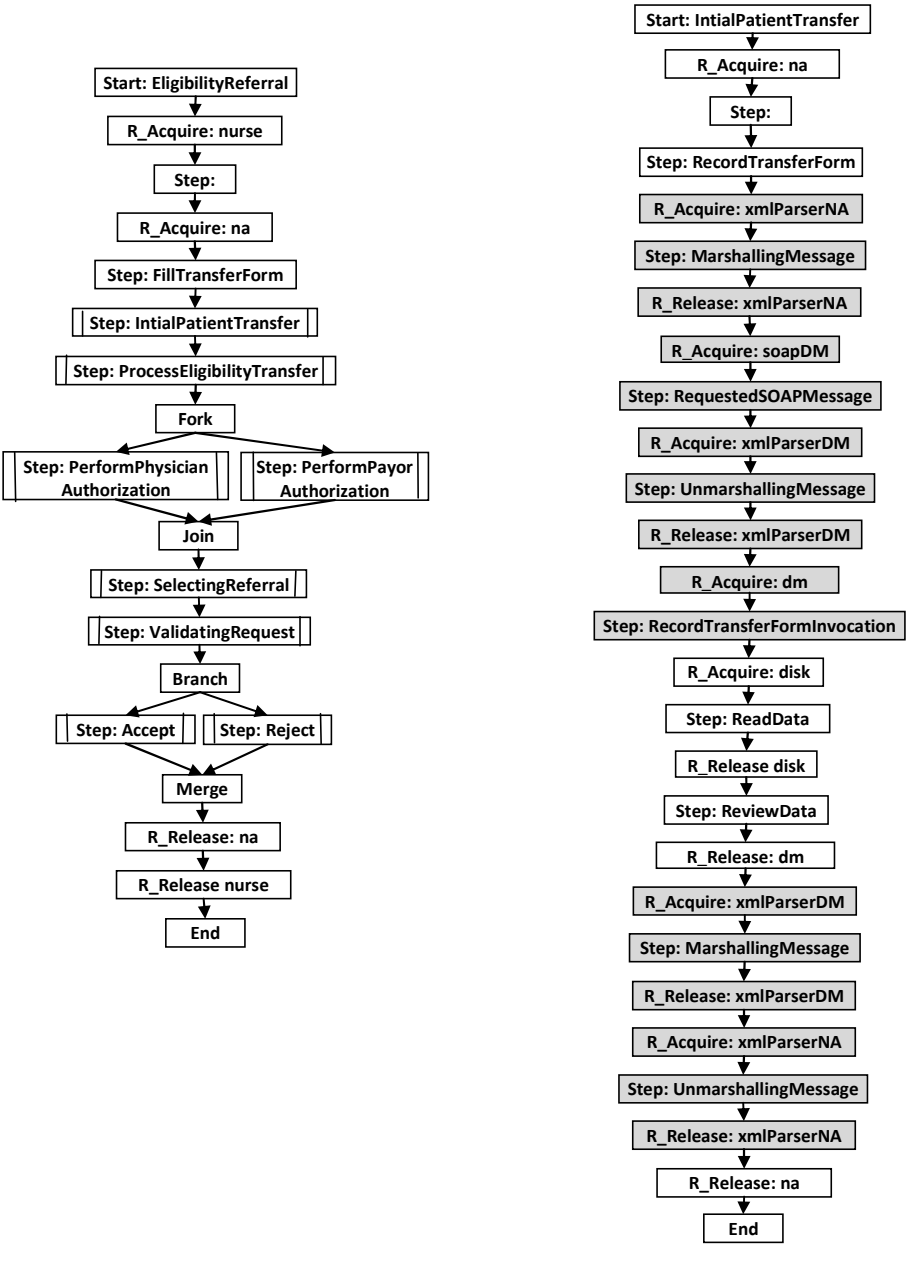

Fig. 17. CSM of the Eligibility Referral System

4.6 From CSM to LQN

Model transformation from CSM to LQN is performed by separating the workflow and service layer, as done in section 3.2. The workflow layer which represents the top level scenario is transformed into an LQN activity graph associated with a task called "workflow", and runs on its own processor. The service layer, which represents CSM sub-scenario containing services, is transformed into a set of tasks with their owned entries corresponding to services. Figure 18 shows the LQN performance model for the Eligibility Referral System. The top level of the LQN represents the workflow activity graph (in gray), while the underlying services are represented by the lower level tasks and entries. The middleware tasks are shown in darker gray.

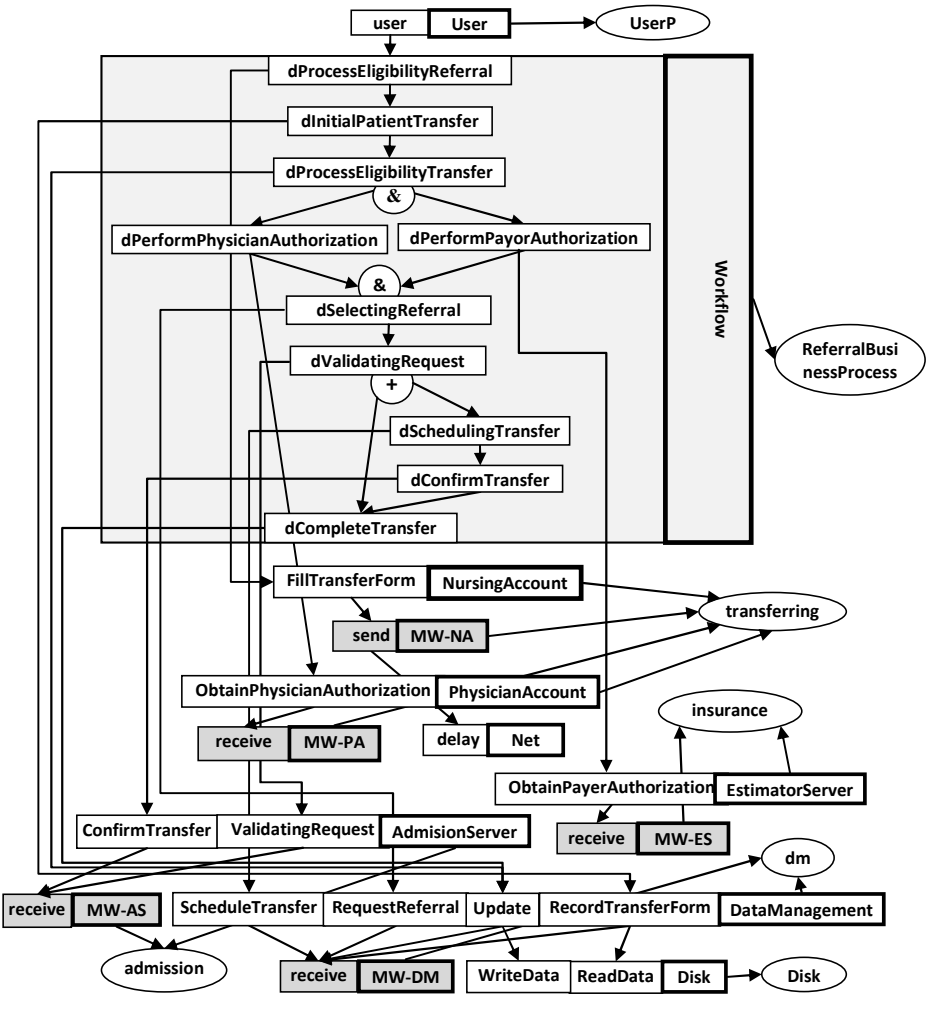

Fig. 18. LQN model

4.7 Performance Results

The performance of the Eligibility Referral System has been evaluated based on two design alternatives: a) with finer service granularity corresponding to service architecture from Figure 11; and b) with coarser service granularity, where the invocations of low level services for accessing the database DM are replaced with regular calls, avoiding the regular service invocation overhead. In the second solution, the functionality of the lower level services have been integrated within the higher level services provided by the NursingAccount component. Figure 18 shows the LQN model generated for the Eligibility Referral System for the first alternative only. The LQN model of the second alternative is not shown here.

The performance analysis is performed to compare the response time and the throughput of the system. It aims to find the system's bottleneck (i.e. software and hardware components that saturate first and throttle the system). To mitigate the bottleneck and improve the performance of the overall system, a series of hardware and/or software modifications are applied after identifying every bottleneck. The LQN results will show the response time reduction obtained by making fewer expensive service invocations using SOAP and XML in the same scenario. Figure 19 compares the response time and throughput of the system versus the number of users ranging from 1 to 100. The results illustrate the difference between the system with finer and coarser granularity. The compared configurations are similar in the number of processors, disks, and threads, except that the latter performs fewer service invocations through the service platform. The improvement is considerable (about 40% for a large number of users).

The results show the importance of service granularity on system performance, which must be evaluated at an early phases of the system design. The proposed analysis helps the modeler to decide on the right granularity level, making a tradeoff between system performance and level of granularity of the deployed services.

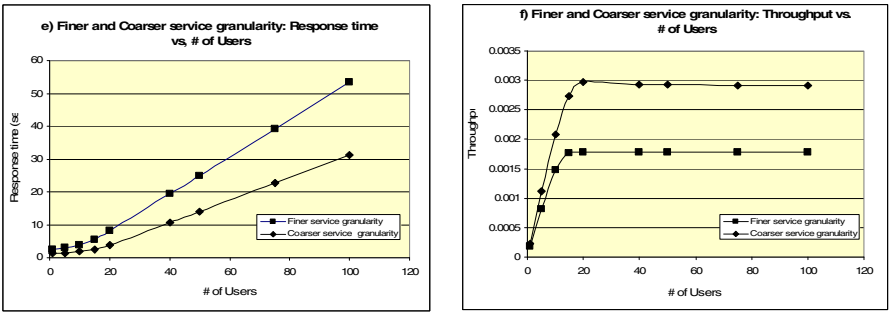

Fig. 19. LQN results for response time and throughput comparing different service granularity

5 Extension of PUMA to Software Product Lines (SPL)

A Software Product Line (SPL) is a set of similar software systems built from a shared set of assets, which are realizing common features satisfying a particular domain. Experience shows that by adopting a SPL development approach, organizations achieve increased quality and significant reductions in cost and time to market [14].

An emerging trend apparent in the recent literature is that the SPL development moves toward adopting a Model-Driven Development (MDD) paradigm. This means that models are increasingly used to represent SPL artifacts, which are building blocks for many different products with all kind of options and alternatives. We propose to integrate performance analysis in the early phases of the model-driven development process for Software Product Lines (SPL), with the goal of evaluating the performance characteristic of different products by generating and analyzing quantitative performance models [39]. Our starting point is the so-called SPL model, a multi-view UML model of the core family assets representing the commonality and variability between different products. We added another dimension to the SPL model, annotating it with generic performance specifications (i.e., using parameters instead of actual values) expressed in the standard UML profile MARTE [30]. Such parameters appear as variables and expression in the MARTE stereotype attributes. A model transformation realized in the Atlas Transformation Language (ATL) derives the UML model of a specific product with concrete MARTE performance annotations from the SPL model. The product derivation process binds the variability expressed in the SPL to a specific product, and also the generic SPL performance annotations to concrete values provided by the designer for this product. The proposed model transformation approach can be applied to any existing SPL model-driven development process using UML for modeling software.

Performance is a runtime property of the deployed system and depends on two types of factors: some are contained in the design model of the product (generated from the SPL model) while others characterize the underlying platforms and runtime environment. Performance models need to reflect both types of factors. Woodside et al. [47] proposed the concept of performance completions to close the gap between abstract design models and external platform factors. Since our goal is to automate the derivation of a performance model for a specific product from the SPL model, we propose to deal with performance completions in the early phases of the SPL development process by using a Performance Completion feature (PC-feature) model as described in the previous section. The PC-feature model explicitly captures the variability in platform choices, execution environments, different types of communication realizations, and other external factors that have an impact on performance, such as different protocols for secure communication channels and represents the dependencies and relationships between them [41]. Therefore, our approach uses two feature models for a SPL: 1) a regular feature model for expressing the variability between member products, and 2) a PC-feature model introduced for performance analysis reasons to capture platform-specific variability.

Dealing manually with a large number of performance parameters and with their mapping, by asking the developers to inspect every diagram in the model, to extract these annotations and to attach them to the corresponding PC-features, is an error-prone process. A model transformation approach is proposed in [43] to automate the collection of all the generic parameters that need to be bound to concrete variables from the annotated product model, presenting them to the user in a user-friendly format.

The automatic derivation of a specific product model based on a given feature configuration is enabled through the mapping between features from the feature model and their realizations in the design model. In this section, an efficient mapping technique is used, which aims to minimize the amount of explicit feature annotations in the UML design model of SPL. Implicit feature mapping is inferred during product derivation from the relationships between annotated and non-annotated model elements as defined in the UML metamodel [40].

In order to analyze the performance of a specific product running on a given platform, we need to generate a performance model for that product by model transformations from the SPL model with generic performance annotations. In our research, this is done in four big steps: a) instantiating a product platform independent model (PIM) with generic performance parameters from the SPL model; b) collecting all the generic parameters that need bounding from the automatically generated product PIM and presents them to the developer in a user-friendly spreadsheet format; c) performing the actual binding to concrete values provided by the developer to obtain a product platform specific model (PSM) and d) generating a performance model for the product from the model obtained in the previous step.

Related Work. To the best of our knowledge, no work has been done to evaluate and predict the performance of a given member of a SPL family by generating a formal performance model. Most of the work aims to model non-functional requirements (NFRs) in the same way as functional requirements. Some of the works are concerned with the interactions between selected features and the NFRs and propose different techniques to represent these interactions and dependencies. In [8], the MARTE profile is analyzed to identify the variability mechanisms of the profile in order to model variability in embedded SPL models. Although MARTE was not defined for product lines, the paper proposes to combine it with existing mechanisms for representing variability, but it does not explain how this can be achieved. A model analysis process for embedded SPL is presented in [9] to validate and verify quality attributes variability. The concept of multilevel and staged feature model is applied by introducing more than one feature models that represent different information at different abstraction levels; however, the traceability links between the multilevel models and the design model are not explained.

In [7], the authors propose an integrated tool-supported approach that considers both qualitative and quantitative quality attributes without imposing hierarchical structural constraints. The integration of SPL quality attributes is addressed by assigning quality attributes to software elements in the solution domain and linking these elements to features. An aggregation function is used to collect the quality attributes depending on the selected features for a given product.

A literature survey on approaches that analyze and design non-functional requirements in a systematic way for SPL is presented in [29]. The main concepts of the surveyed approaches are based on the interactions between the functional and non-functional features.

An approach called Svamp is proposed to model functional and quality variability at the architectural level of the SPL [35]. The approach integrates several models: a Kumbang model to represent the functional and structural variability in the architecture and to define components that are used by other models; a quality attribute model to specify the quality properties and a quality variability model for expressing variability within these quality attributes.

Reference [10] extends the feature model with so-called extra-functional features representing non-functional features. Constraint programming is used to reason on this extended feature model to answer some questions such as how many potential products the feature model contains.

The Product Line UML-Based Software Engineering (PLUS) method is extended in [38] to specify performance requirements by introducing several stereotypes specific to model performance requirements such as «optional» and «alternative performance feature».

5.1 Domain Engineering Process

The SPL development process is separated into two major phases: 1) *domain engineering* for creating and maintaining a set of reusable artifacts and introducing variability in these software artifacts, so that the next phase can make a specific decision according to the product's requirements; and 2) *application engineering* for building family member products from reusable artifacts created in the first phase instead of starting from scratch.

The domain engineering process is a development cycle *for* reuse and includes, but is not limited to, creating the requirement specifications, domain models, architecture, reusable software components [14]. The SPL assets created by the domain engineering process which are of interest for our research are represented by a multi-view UML design model of the family, called the *SPL model*, which represents a superimposition of all variant products. The creation of the SPL model employs two separate UML profiles: a *product line* profile based on [24] for specifying the commonality and variability between products, and the MARTE profile for performance annotations. Another important outcome of the domain engineering process is the feature model used to represent commonalities and variabilities between family members in a concise taxonomic form. Additionally, the PC-feature model is created to represent the variability space of the performance completions.

An e-commerce case study is used to illustrate the construction of the UML model for SPL that represents the source model of our model transformation approach. The e-commerce SPL is a web-based product line that can generate a distributed application that can handle either business-to-business (B2B) or business-to-consumer (B2C) systems. For instance, in B2B, a business customer can browse and select items through several catalogs. Each customer has a contract with a supplier for purchases, as well as bank accounts through which payments can be made. An operation fund is associated with each contract.

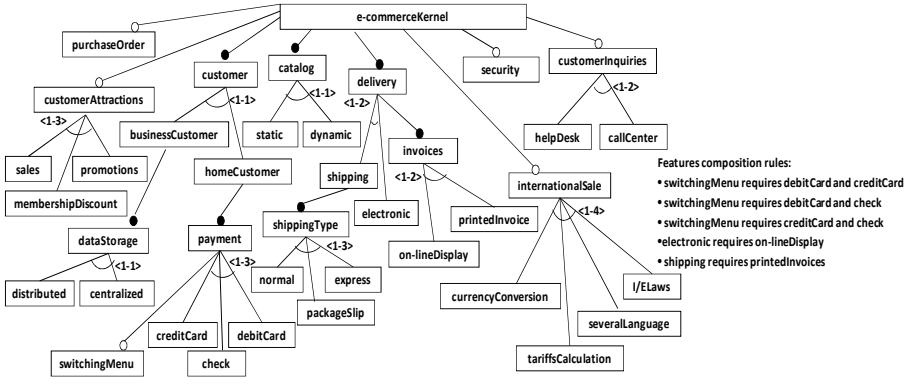

Fig. 20. Feature model of the e-commerce SPL

Feature Model. The feature models are used in our approach to represent two different variability spaces. The PC-feature model represents the variability in platform choices, execution environments, and other external factors that have an impact on performance as described in the previous section. This sub-section describes the regular feature model representing functional variabilities between products. An example of feature model of an e-commerce SPL is represented in Figure 20 in the extended FODA notation, Cardinality-Based Feature Model (CBFM) [17]. Since the FODA notation is not part of UML, the feature diagram is represented in the source model taken as input by our ATL transformation as an extended UML class diagram, where the features and feature groups are modeled as stereotyped classes and the dependencies and constraints between features as stereotyped associations. For instance, the two alternative features *Static* and *Dynamic* are mutually exclusive and so they are grouped into an *exactly-one-of* feature group called *Catalog*, while the three optional features CreditCard, DebitCard, and Check are grouped into an at-least-one-of feature group called Payment. Thus, an individual system can provide at least one of these features or any number of them. In the case of an individual system providing all of these features, the user can choose one of them during the run-time execution. In addition to functional features, we add to the diagram another type of features characterizing design decisions that have an impact on the non-functional requirements or properties. For example, the architectural decision related to the location of the data storage (centralized or distributed) affects performance, reliability and security, and is represented in the diagram by two mutually exclusive quality features. This type of feature related to a design decision is part of the design model, not just an additional PC-feature required only for performance analysis. This feature model represents the set of all possible combinations of features for the products of the family. It describes the way features can be combined within this SPL. A specific product is configured by selecting a valid feature combination from the feature model, producing a so-called feature configuration based on the product's requirements. To enable the automatic derivation of a given product model from the SPL model, the mapping between the features

contained in the feature model and their realizations in a reusable SPL model needs to be specified, as shown in the next sub-section. Also, each stereotyped class in the feature model has a tagged value indicating whether it is selected in a given feature configuration or not.

SPL Model. The SPL model should contain, among other assets, structural and behavioural views which are essential for the derivation of performance models. It consists of: 1) structural description of the software showing the high-level classes or components, especially if they are distributed and/or concurrent; 2) deployment of software to hardware devices; 3) a set of key performance scenarios defining the main system functions frequently executed.

The functional requirements of the SPL are modeled as use cases. Use cases required by all family members are stereotyped as *«kernel»*. The variability distinguishing the members of a family from each other is explicitly modeled by use cases stereotyped as *«optional»* or *«alternative»*. In order to avoid polluting our model with extra annotations and to ensure the well-formedness of the derived product model, we propose to annotate explicitly the minimum number of model elements within each diagram of our SPL model. For instance, in the use case diagram, only the optional and alternative use cases are annotated with the name of the features requiring them (given as stereotype attributes); since a kernel use case represents commonality, it is sufficient to just stereotype it as «kernel». Other model elements, such as actors, associations, generalizations, properties, are mapped implicitly to feature through their relationship with the use cases, so there is no need to clutter the model with their annotations. The evaluation of implicit mapping during product derivation is explained in the following subsection.

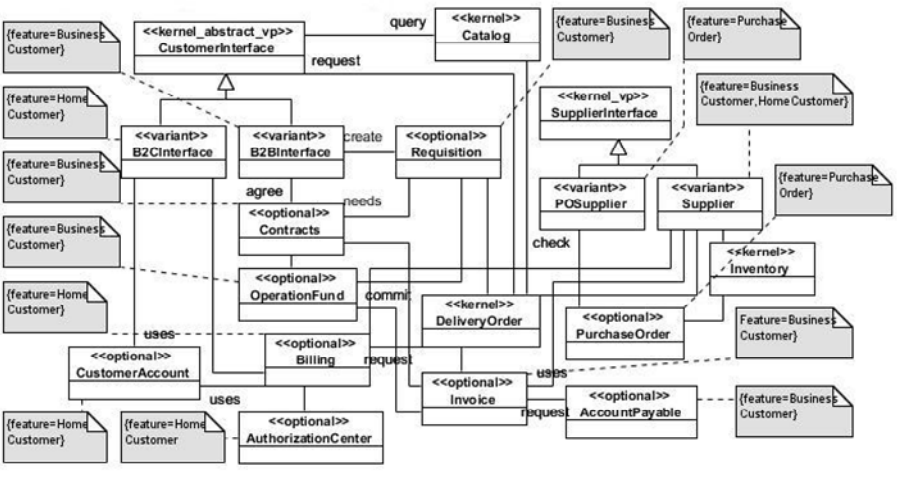

Fig. 21. A fragment of the class diagram of the e-commerce SPL

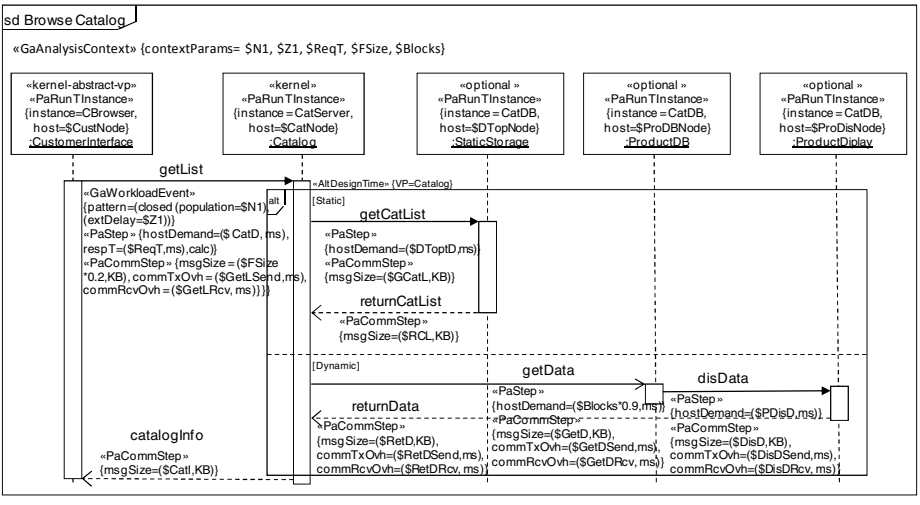

Fig. 22. SPL Scenario Browse Catalog

The structural view of the SPL is presented as a class diagram; Figure 21 depicts a small fragment. The classes that are common to all members of the SPL are stereotyped as *«kernel»*. The variability that distinguishes the members of a family from each other is explicitly modeled by classes stereotyped as *«optional»* or annotated with the name of the feature(s) requiring them (given as stereotype attributes). This is an example of mapping between features and the model elements realizing them. In cases where a class behaves differently in different product (such as CustomerInterface in B2B and B2C systems) a generalization/specialization hierarchy is used to model the different behaviours of this class. The two subclasses B2BInterface and B2CInterface are used by B2B systems and B2C systems, respectively.

The behavioural SPL view is modeled as sequence diagrams for each scenario of each use case of interest. Figure 22 illustrates the kernel scenario *BrowseCatalog*. Sequence diagram variability that distinguishes between the behaviour of different features is expressed by extending the *alt* and *opt* fragments with the stereotypes *«AltDesignTime» «OptDesignTime»*, respectively. For example, the *alt* fragment stereotyped with *«AltDesignTime» {VP=Catalog}* gives two choices based on the value of the *Catalog* feature (*Static* or *Dynamic*); more specifically, each one of its *Interaction Operand* has a guard denoting the feature *Static* or *Dynamic*. An alternative feature that is rather complex and is represented as an extending use case, can be also modeled as an extended *alt* operator that contains an *Interaction Use* referring to an *Interaction* representing the extending use case. Note that regular *alt* and *opt* fragments that are not stereotyped represent choices to be made at runtime, as defined in UML.

Since the SPL model is generic, covering many products and containing variation points with variants, the MARTE annotations need to be generic as well. We use

MARTE variables as a means of parameterizing the SPL performance annotations; such variables (parameters) will be assigned (bound to) concrete values during the product derivation process. For instance the message *getList* is stereotyped as a communication step (by convention, we use names starting with '$' for all MARTE variables to distinguish them from other identifiers and names):

«PaCommStep» { msgSize = ($MReq, KB),
commTxOvh = ($GetLSend, ms),
commRcvOvh = ($GetLRcv, ms)}}

where the message size is the variable *$GetL* in KiloBytes. The overheads for sending and receiving this particular message are the variables *$GetLSend* and *$GetLRcv*, respectively, in milliseconds. We propose to annotate each communication step (which corresponds to a logical communication channel) with the CPU overheads for transferring the respective message: *commTxOvh* for transmitting (sending) the message and *commRcvOvh* for receiving it. Eventually, these overheads will be added in the performance model to the execution demands of the two execution hosts involved in the communication (one for sending and the other for receiving the respective message).

Performance Completions. In SPL, different members may vary from each other in terms of their functional requirements, quality attributes, platform choices, network connections, physical configurations, and middleware. Many details contained in the system that are not part of its design model but of the underlying platforms and environment, do affect the run-time performance and need to be represented in the performance model. *Performance completions*, as proposed by Woodside [47] and explained in the previous section, are a manner to close the gap between the high-level design model and its different implementations. Performance completions provide a general concept to include low-level details of execution environment/ platform in performance models

In this approach, we propose to include the performance impact of underlying platforms into the UML+MARTE model of a product as aggregated platform overheads, expressed in MARTE annotations attached to existing processing and communication resources in the generated product model. This will keep the model simple and still allow us to generate a performance model containing the performance effects of both the product and the platforms. Every possible PC-feature choice is mapped to certain MARTE annotations corresponding to UML model elements in the product model. This mapping is realized by the transformation generating the parameter spreadsheets, which is providing the user with mapping information in order to put the annotation parameters needing to be bound to concrete values into context.

Adding security solutions requires more resources and longer execution times, which in turn has a significant impact on system performance. The PC-feature group *Communication* shown in Figure 13 contains two alternative features *secured* and *unsecured*. The *secured* feature offers two security protocols, each with different overheads for sending and receiving secure messages. These overheads are mapped to the communication overheads through the attributes *commRcvOvh* and *commTxOvh*,

which represent the host demand overheads for receiving and sending messages, respectively. Since not all the messages exchanged in a product need to have the same communication overheads, we propose to annotate each individual message stereotyped as *«PaCommStep»* with the processing overheads for the respective message: *commTxOvh* for transmitting (sending) it and *commRcvOvh* for receiving it. In fact, these overheads correspond to the logical communication channel that conveys the respective message. Eventually, the logical channel will be allocated to a physical communication channel (e.g., network or bus) and to two execution hosts, the sender and the receiver. The *commTxOvh* overhead will be eventually added in the performance model to the execution demands of the sender host and *commRcvOvh* to that of the receiver host.

Each feature from the PC-feature model shown in Figure 13 may affect one or more performance attributes. For instance, data compression reduces the message size and at the same time increases the processor communication overhead for compressing and decompressing the data. Thus, it is mapped to the performance attributes message size and communication overhead through the MARTE attributes *msgSize*, *commTxOvh* and *commRcvOvh*, respectively. The mapping here is between a PC-feature and the performance attribute(s) affected by it, which are represented as MARTE stereotype attributes associated to different model elements.

5.2 Model Transformation Approach

The derivation of a specific UML product model with concrete performance annotations from the SPL model with generic annotations requires three model transformations: a) transforming the SPL model to a product platform independent model (PIM) with generic performance annotations, b) generating spreadsheets for the user containing generic parameters and guiding information for the specific product, c) performing the actual binding by using the concrete values provided by the user to produce a product platform specific model (PSM). We have implemented these model transformations in the Atlas Transformation Language (ATL) [1]. We handle two kind of generic parametric annotations: a) product-specific (due to the variability expressed in the SPL model) and platform-specific (due to device choices, network connections, middleware, and runtime environment).

Product PIM Derivation. The derivation process is initiated by specifying a given product through its feature configuration (i.e., the legal combination of features characterizing the product). The selected features are checked for consistency against the feature dependencies and constraints in the feature model, in order to identify any inconsistencies. An example is checking to ensure that no two mutually exclusive features are chosen.

The second step in the derivation process is to select the use cases realizing the chosen features. All kernel use cases are copied to the product use case diagram, since they represent functionality provided by every member of the SPL. If a chosen feature is realized through extend or include relationships between use cases, both the base and the included or extending use cases have to be selected, as well. A use case

containing in its scenario variation point(s) required to realize the selected feature(s) has to be chosen, too. The optional and alternative use cases are selected and copied to the target use case diagram if they are mapped to a feature from the feature configuration. The implicit mapping of other non-annotated elements is inferred from their relationships with annotated elements as defined in the UML metamodel and well-formedness rules. For example, Actor is a non-annotated element associated to one or more use cases, so its implicit mapping is evaluated through the attribute *memberEnd* owned by the *Association* connected it with a use case. The attribute *memberEnd* collects all the properties related to the association and since the *type* of the property refers to the end of the association, we can navigate to the use case and the corresponding actor through this attribute. Whenever, the use case is selected, the actor and the association are selected as well. Finally, the use case diagram for the product is developed after all the PL variability stereotypes were eliminated.

The third step is to derive the product class diagram by selecting first all kernel classes from the SPL class diagram. Optional and variant classes needed for the desired product are selected next (each is annotated with the feature(s) requiring it). Moreover, superclasses of the selected optional or variant classes have to be selected as well. The other non-annotated elements are selected based on their relationships with annotated elements as defined in the UML metamodel. For example, according to the UML metamodel, a binary association has to be attached to a classifier at each end. Therefore, the decision whether a binary association has to be copied or not to the target is based on the selection of both of its classifiers. If at least one of the classifiers is not selected, the association will not be created in the target model. In other words, the binary association is created in the target model if and only if both of its *memberEnd* properties have their classifiers already selected and created. At the same time, if only one of its classifier is selected, the property attached to this unselected association and owned by the selected classifier should not be created in the target model.

The final step of the product derivation is to generate the sequence diagrams corresponding to different scenarios of the chosen use cases. Each such scenario is modeled as a sequence diagram, which has to be selected from the SPL model and copied to the product one. The PL variability stereotypes are eliminated after binding the generic roles associated to the lifelines of each selected sequence diagram to specific roles corresponding to the chosen features. For instance, the sequence diagram *BrowseCatalog* has the generic alternate role *CustomerInterface* which has to be bound to a concrete role, either *B2BInterface* or *B2CInterface* to realize the features *BusinessCustomer* or *HomeCustomer*, respectively. However, the selection of the optional roles is based on the corresponding features. For instance, the generic optional role *StaticStorage* is selected if the feature *Static Catalog* is chosen. More details about the derivation approach and the mapping of functional features to model elements are presented in our previous work [40] [42].

The outcome of this model transformation is a product model where the variability related to SPL has been resolved based on the chosen feature configuration. However, the performance annotations are still generic and need to be bound to concrete values.

Generating User-Friendly Representation. The generic parameters of a product PIM derived from the SPL model are related to different kind of information: a) product-specific resource demands (such as execution times, number of repetitions and probabilities of different steps); b) software-to-hardware allocation (such as component instances to processors); and c) platform/environment-specific performance details (also called performance completions). The user (i.e., performance analyst) needs to provide concrete values for all generic parameters; this will transform the generic product model into a platform-specific model describing the run-time behaviour of the product for a specific run-time environment.

Choosing concrete values to be assigned to the generic performance parameters of type (a) is not a simple problem. In general, it is difficult to estimate quantitative resource demands for each step in the design phase, when an implementation does not exist and cannot be measured yet. Several approaches are used by performance analysts to come up with reasonable estimates in the early design stages: expert experience with previous versions or with similar software, understanding of the algorithm complexity, measurements of reused software, measurements of existing libraries, or using time budgets. As the project advances, early estimates can be replaced with measured values for the most critical parts. Therefore, it is helpful for the user of our approach to keep a clearly organized record for the concrete values used for binding in different stages of the project. For this reason, we proposed to automate the collection of the generic parameters from the model on spreadsheets, which will be provided to the user.

The parameters of type (b) are related to the allocation of software components to processors available for the application. The user has to decide for a product what the actual hardware configuration is and how to allocate the software to processing nodes. The MARTE stereotype *«RunTInstance»* annotating a lifeline in a sequence diagram provides an explicit connection between a role in the behaviour model and the corresponding runtime instance of a component. The attribute host of this stereotype indicates on which physical node from the deployment diagram the instance is running. Using parameters for the attribute host enable us to allocate each role (a software component) to an actual hardware resource. The transformation collects all these hardware resources and associates their list to each lifeline in the spreadsheets. The user decides on the actual allocation by choosing a processor from this list.

The performance effects of variations in the platform/environment factors (such as network connections, middleware, operating system and platform choices) are included into our model by aggregating the overheads caused by each factor and by attaching them via MARTE annotations to the affected model elements. As already mentioned, the variations in platform/environment factors are represented in our approach through the PC-feature model (as explained in the previous section). A specific run-time instance of a product is configured by selecting a valid PC-feature combination from the PC-feature model. We define a PC-feature configuration as a complete set of choices of PC-features for a specific model element.

It is interesting to note that a PC-feature has impact on a subset of model elements in the model, but not necessarily on all model elements of the same type. For instance, the PC-feature *Secured* affects only certain communication channels in a product

Element Type	Element Name	Stereotype Name	AttributeName	PC-Feature Group Name	PC-Feature Name	Guideline for Value	Generic Parameter	Concrete Value
Context Analysis Parameters {$N1, $Z1, $ReqT, $FSize, $Blocks}								
Message	getList	PaStep	hostDemand	application-annotation			$CatD (ms)	
		PaCommStep	msgSize	«exactly-one-of feature» DataCompression	Compressed	reduce by 10% ...30%		
					Uncompressed	No effect		
							$FSize *0.2 (KB)	
			commTxOverhead	«exactly-one-of feature» Communication	unsecured	No effect		
					secured			
					SSL	add (12.5+0.078*msgsize)		
					TLS	add (11.9+0.134*msgsize)		
				«exactly-one-of feature» DataCompression	compressed	increase by 2% ...5%		
					uncompressed	No effect		
							$GetLSend (ms)	
			commRcvOverhead	«exactly-one-of feature» Communication	unsecured	No effect		
					secured			
				«exactly-one-of feature» securityProtocol	SSL	add (12.5+0.078*msgsize)		
					TLS	add (11.9+0.134*msgsize)		
				«exactly-one-of feature» DataCompression	compressed	increase by 2% ...5%		
					uncompressed	No effect		
							$GetLRcv (ms)	

Fig. 23. Part of the generated Spreadsheet for the scenario Browse Catalog

model, not all of them. Hence, a PC-feature needs to be associated to certain model element(s), not to the entire product. This mapping is set up through the MARTE performance specifications annotating the affected model elements.

Dealing manually with a huge number of performance annotations by asking the developer to inspect every diagram in the generated product model, to extract the generic parameters and to match them with the PC-features is an error-prone process. We propose to automate the process of collecting all generic parameters that need to be bound to concrete values from the product model and to associate each PC-feature to the model element(s) it may affect, then present the information to the developer in a user-friendly format. We generate a spreadsheet per diagram, indicating for each generic parameter some guiding information that helps the user in providing concrete binding values.

The transformation handles differently the context analysis parameters, which are usually defined by the modeler to be carried without binding throughout the entire transformation process, from the SPL model to the performance model for a product. These parameters can be used to explore the performance analysis space. A list of the context analysis parameters are provided to the user, who will decide whether to bind them now to concrete values, or to use them unbound in MARTE expressions.

A part of the generated spreadsheet for the scenario *BrowseCatalog* is shown in Figure 23. For instance, the PC-feature *DataCompression* is mapped to the MARTE attribute *msgSize* annotating a model element of type message. As the value of the attribute *msgSize* is an expression *$FSize*0.2* in function of the context analysis parameter *$FSize*, it is the user's choice to bind it at this level or keep it as a parameter in the output it produces. The column titled Concrete Value is designated for the user to enter appropriate concrete value for each generic parameter, while the column Guideline for Value provides a typical range of values to guide the user. For instance, if the PC-selection features chosen are "secured" with "TLS", the concrete value entered by the user is obtained by evaluating the expression (11.9+0.134*msgsize), assuming

that the user follows the provided guideline. Assuming that the choice for the PC-feature *DataCompression* is "compressed", the user may decide to increase by 4% the existing overhead due to compression features. In general, the guidelines can be adjusted by the performance analyst for a given SPL and a known execution environment. The generated spreadsheet presents a user-friendly format for the users of the transformation who have to provide appropriate concrete values for binding the generic performance annotations. Being automatically generated, they capture all the parameters that need to be bound and reduce the incidence of errors.

Performing the Actual Binding. After the user selects an actual processor for each lifeline role provided in the spreadsheets and enters concrete values for all the generic performance parameters, the next model transformation takes as input these spreadsheets along with its corresponding product model, and binds all the generic parameters to the actual values provided by the user. The outcome of the transformation is a specific product model with concrete performance annotations, which can be further transformed into a performance model.

In order to automate the actual binding process, the generated spreadsheets with concrete values are given as a mark model to the binding transformation. The mark model concept has been introduced in the OMG MDA guide [33] as a means of providing concrete parameter values to a transformation. This capability of allowing transformation parameterization through mark model instances makes the transformation generic and more reusable in different contexts.

6 Conclusions

In this chapter we presented the open PUMA tool architecture that can accept a variety of types of Smodels and generate a variety of types of Pmodels. The practicality of PUMA is demonstrated by different implemented transformations from UML 1.4 and UML 2.X to CSM for sequence and activity diagrams, and transformations from CSM to queueing networks, LQN and Petri nets. We are extending PUMA for SOA and SPL and are working on the final component of PUMA, to support the systematic use of performance models in order to generate feedback to the designers. PUMA promises a way out of the maze of possible evaluation techniques. From the point of view of practical adoption, this is of the utmost importance, as the software developer is not tied to an evaluation model whose limitations he or she does not understand. Performance modelers are similarly freed to generate a wide variety of forms of model, and explore their relative capabilities, without having to create the (quite difficult) interface to UML. As UML is constantly changing, this can also make maintenance of model-building easier. While PUMA is described for performance, CSM may be adapted to other evaluations based on behaviour.

In general, experience in conducting model-driven performance analysis and other non-functional properties (NFPs) in the context of model-driven development 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 performance and other kind of NFPs, which leads to the idea of hiding 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 or platform versions.

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 work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) and industrial and government partners, through the Healthcare Support through Information Technology Enhancements (hSITE) Strategic Research Network and through Discovery grants.

References

[1] Abdul Fatah, I., Majumdar, S.: Performance of CORBA-Based Client Server Architectures. IEEE Transactions on Parallel & Distributed Systems, 111–127 (February 2002)

[2] Alhaj, M., Petriu, D.C.: Approach for generating performance models from UML models of SOA systems. In: Proceedings of CASCON 2010, Toronto, November 1-4 (2010)

[3] Anyanwu, K., Sheth, A., Cardoso, J., Miller, J., Kochut, K.: Healthcare Enterprise Process Development and Integration. Journal of Research and Practice in Information Technology 35(2) (May 2003)

[4] Atlas Transformation Language (ATL), `http://www.eclipse.org/m2m/atl`

[5] Balsamo, S., DiMarco, A., Inverardi, P., Simeoni, M.: Model-based Performance Prediction in Software Development. IEEE Transactions on Software Eng. 30(5), 295–310 (2004)

[6] Balsamo, S., Marzolla, M.: Simulation Modeling of UML Software Architectures. In: Proc. ESM 2003, Nottingham, UK (June 2003)

[7] Bartholdt, J., Medak, M., Oberhauser, R.: Integrating Quality Modeling with Feature Modeling in Software Product Lines. In: Proc. of the 4th Int. Conference on Software Engineering Advances (ICSEA 2009), pp. 365–370 (2009)
[8] Belategi, L., Sagardui, G., Etxeberria, L.: MARTE Mechanisms to Model Variability When Analyzing Embedded Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 466–470. Springer, Heidelberg (2010)
[9] Belategi, L., Sagardui, G., Etxeberria, L.: Model based analysis process for embedded software product lines. In: Proc. of 2011 Int. Conference on Software and Systems Process, ICSSP 2011 (2011)
[10] Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
[11] 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, pp. 35–45 (July 2002)
[12] Bernardi, S., Merseguer, J.: Performance evaluation of UML design with Stochastic Well-formed Nets. Journal of Systems and Software 80(11), 1843–1865 (2007)
[13] Cavenet, C.G., Hillston, J., Kloul, L., 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, pp. 74–83 (January 2004)
[14] Clements, P.C., Northrop, L.M.: Software Product Lines: Practice and Patterns, p. 608. Addison-Wesley (2001)
[15] Cortellessa, V., Di Marco, A., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)
[16] Cortellessa, V., Mirandola, R.: Deriving a Queueing Network based Performance Model from UML Diagrams. In: Proc. Second Int. Workshop on Software and Performance, Ottawa, September 17-20, pp. 58–70 (2000)
[17] Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice, 7–29 (2005)
[18] D'Ambrogio, A., Bocciarelli, P.: A Model-driven Approach to Describe and Predict the Performance of Composite Services. In: WOSP 2007, Buenos- Aires, Argentina (2007)
[19] Earl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education (2005)
[20] DiStefano, S., Scarpa, M., Puliafito, A.: From UML to Petri Nets: The PCM-Based Methodology. IEEE Trans. on Software Engineering 37(1), 65–79 (2011)
[21] France, R., Ray, I., Georg, G., Ghosh, S.: An Aspect-Oriented Approach to Early Design Modeling. In: IEE Proceedings - Software, Special Issue on Early Aspects (2004)
[22] Franks, G., Hubbard, A., Majumdar, S., Petriu, D.C., Rolia, J., Woodside, C.M.: A toolset for Performance Engineering and Software Design of Client-Server Systems. Performance Evaluation 24(1-2), 117–135 (1995)
[23] Franks, G.: Performance Analysis of Distributed Server Systems, Report OCIEE-00-01, Ph.D. Thesis, Carleton University, Ottawa, Canada (2000)
[24] Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. Addison-Wesley Object Technology Series (July 2005)
[25] Gómez-Martínez, E., Merseguer, J.: Impact of SOAP Implementations in the Performance of a Web Service-Based Application. In: Min, G., Di Martino, B., Yang, L.T., Guo, M., Rünger, G. (eds.) ISPA Workshops 2006. LNCS, vol. 4331, pp. 884–896. Springer, Heidelberg (2006)

[26] Grassi, V., Mirandola, R., Randazzo, E., Sabetta, A.: KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 327–356. Springer, Heidelberg (2008)
[27] Happe, J., Becker, S., Rathfelder, C., Friedrich, H., Reussner, R.: Parametric performance completions for model-driven performance prediction. Performance Evaluation 67(8), 694–716 (2010)
[28] Marzolla, M., Mirandola, R.: Performance Prediction of Web Service Workflows. In: Overhage, S., Ren, X.-M., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 127–144. Springer, Heidelberg (2008)
[29] Nguyen, Q.: Non-Functional Requirements Analysis Modeling for Software Product Lines. In: Proc. of Modeling in Software Engineering (MISE 2009), ICSE Workshop, pp. 56–61 (2009)
[30] Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), Version 1.1, OMG document formal/2011-06-02 (2011)
[31] Object Management Group, UML Profile for Schedulability, Performance, and Time Specification, Version 1.1, OMG document formal/05-01-02 (January 2005)
[32] Object Management Group, Service oriented architecture Modeling Language (SoaML), ptc/2009-04-01 (April 2009)
[33] Object Management Group, MDA Guide Version 1.0.1, omg/03-06-01 (2003)
[34] Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and Systems Modeling 6(2), 163–184 (2007)
[35] Raatikainen, M., Niemelä, E., Myllärniemi, V., Männistö, T.: Svamp - An Integrated Approach for Modeling Functional and Quality Variability. In: 2nd Int Workshop on Variability Modeling of Software-intensive Systems, VaMoS (2008)
[36] Rolia, J.A., Sevcik, K.C.: The Method of Layers. IEEE Trans. on Software Engineering 21(8), 689–700 (1995)
[37] Smith, C.U.: Performance Engineering of Software Systems. Addison Wesley (1990)
[38] Street, J., Gomaa, H.: An Approach to Performance Modeling of Software Product Lines. In: Workshop on Modeling and Analysis of Real-Time and Embedded Systems, Genova, Italy (October 2006)
[39] Tawhid, R., Petriu, D.C.: Integrating Performance Analysis in the Model Driven Development of Software Product Lines. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 490–504. Springer, Heidelberg (2008)
[40] Tawhid, R., Petriu, D.C.: Product Model Derivation by Model Transformation in Software Product Lines. In: Proc. of the 2nd IEEE Workshop on Model-based Engineering for Real-Time Embedded Systems (MoBE-RTES 2011), Newport Beach, CA, USA (2011)
[41] Tawhid, R., Petriu, D.C.: Automatic Derivation of a Product Performance Model from a Software Product Line Model. In: Proc. of the 15th International Conference on Software Product Line (SPLC 2011), Munich, Germany (2011)
[42] Tawhid, R., Petriu, D.C.: Integrating Performance Analysis in Software Product Line Development Process. In: Software Product Lines - The Automated Analysis. InTech - Open Access Publisher (2011)
[43] Tawhid, R., Petriu, D.C.: User-Friendly Approach for Handling Performance Parameters during Predictive Software Performance Engineering. In: Proc. of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012), Boston, USA (2012)

[44] Tribastone, M., Gilmore, S.: Automatic Translation of UML Sequence Diagrams into PEPA Models. In: Proc. of 5th Int. Conference on Quantitative Evaluation of SysTems (QEST 2008), St Malo, France, pp. 205–214 (2008)

[45] Verdickt, T., Dhoedt, B., Gielen, F., Demeester, P.: Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models. IEEE Trans. on Software Eng. 31(8), 695–711 (2005)

[46] Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S.: The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. IEEE Transactions on Computers 44(1), 20–34 (1995)

[47] Woodside, C.M., Petriu, D.C., Siddiqui, K.H.: Performance-related Completions for Software Specifications. In: Proc. of the 22nd Int. Conference on Software Engineering, ICSE 2002, Orlando, Florida, USA, pp. 22–32 (2002)

[48] Woodside, C.M., Petriu, D.C., Petriu, D.B., Xu, J., Israr, T., Georg, G., France, R., Houmb, S.H., Jürjens, J.: Performance Analysis of Security Aspects by Weaving Scenarios Extracted from UML Models. Journal of Systems and Software 82, 56–74 (2009)

[49] Xu, J.: Rule-based automatic software performance diagnosis and improvement. Performance Evaluation 67(8), 585–611 (2010)