Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links

Taghreed Altamimi, Mana Hassanzadeh Zargari, Dorina Petriu
Carleton University
Systems and Computer Engineering Department
Ottawa, Ontario, Canada
{taghreedaltamimi | manazargar | petriu}@sce.carleton.ca

ABSTRACT

This paper proposes an approach for performance analysis roundtrip in the context of model-driven engineering (MDE) of real-time distributed and embedded systems. The starting point is a UML software model with MARTE performance annotations, such as performance requirements and resource demands. The source software model is automatically transformed into a Layered Queueing Network (LQN) performance model. We developed the transformation with Epsilon, a family of languages for model-to-model transformation, model validation and model management. Using specialized languages helped us create a more compact transformation, easier to understand and maintain than transformations developed with general purpose languages, such as Java. Beside the performance model, the transformation also generates a traceability model containing trace links between mapped elements of the software and performance model. After solving the performance model with an existing solver, the performance results are fed back to the software model by following in reverse the cross-model trace links. The software developers can see the performance results as MARTE stereotype attributes, using a standard UML editor. The approach is illustrated by applying it to an e-commerce application.

CCS Concepts

• Software and its engineerin~~g~~ Model -driven software engineering • Software and its engineerin~~g~~ Software performance

Keywords

performance analysis; model transformation; trace links; UML; MARTE; LQN; Epsilon.

1. INTRODUCTION

Software performance engineering (SPE) is a systematic quantitative approach to construct software systems that meet their performance requirements. It is based on the careful and methodical assessment of software performance properties throughout the software lifecycle, from requirements and design to implementation and maintenance [23][24].

CASCON'16, October 31– November 2, 2016, Toronto, ON, Canada.
Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.
DOI: http://dx.doi.org/10.1145/12345.67890

SPE provides developers with quantitative performance results, such as throughput, response time and utilization, obtained from solving the performance models that it produces from the earliest software development phases. The goal is to allow developers to assess as early as possible the performance effect of different architecture, design, implementation and deployment alternatives, in order to satisfy the performance requirements [3][11].

In order to help the developers to understand and interpret the performance results from the point of view of the software rather than performance model, this paper proposes an approach for performance analysis roundtrip in the context of model-driven engineering (MDE) of real-time distributed and embedded systems, as shown in Figure 1. The starting node S represents a UML software model with MARTE performance annotations that is transformed into a Layered Queueing Network (LQN) [13] performance model represented by node P. The transformation TransS2P was developed with Epsilon (standing for "Extensible Platform of Integrated Languages for model management") a family of languages specialized for model to model trans-formation, model validation and model management [15]. Using Epsilon Transformation Language (ETL) facilitates the automatic generation of cross-model trace links along with the generation of LQN target elements in one-step transformation. Cross model traceability means having direct trace links between S and P, which may help in different ways: a) propagate small changes from S to P, b) support the co-evolution of the software and performance model, and c) import the performance results obtained from solving P to the software domain. Point (c) is discussed in this paper, while (a) and (b) are left for future work. **P'** represents the LQN model with performance results after solving **P** with an existing solver, and S' is the software model with performance results stored as values of MARTE stereotype attributes. The trace links can be examined in the reverse way,

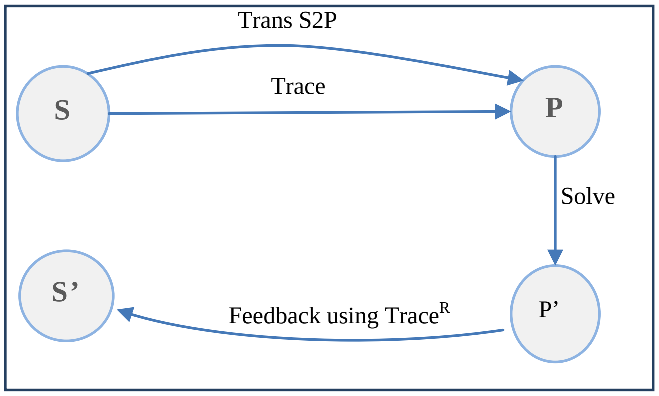


Figure 1. Performance Analysis Roundtrip

starting from elements in **P'** to feed back the performance results to the corresponding elements in **S'**.

This paper has three objectives: 1) presenting the light-weight TransS2P transformation developed in Epsilon, which has more powerful means of expressing transformation rules than non-specialized languages; 2) automating the generation of cross model traces; and 3) presenting the performance analysis roundtrip to feedback performance results to the UML+MARTE model. The proposed roundtrip can also be used with other analysis models for nonfunctional requirements, such as reliability, availability, security and safety.

The paper is organized as follows. Section 2 provides an overview of related work in this research area. Section 3 the source and target models, illustrated with a running example of an e-commerce system. Section 4 discusses the transformation process. Section 5 describes the procedure of feeding back the performance results to the software model. Section 6 applies the proposed approach to the e-commerce example. The last section concludes the paper.

2. RELATED WORK

In the software performance engineering field, there have been significant efforts to integrate performance analysis into the software development process by using different performance modeling formalisms: queueing networks, Petri nets, stochastic process algebras, and simulation. A good survey of the techniques for deriving performance models from UML models is given in [4] and later in the book [7]. A few early examples of derivation of different kinds of performance models from UML are as follows. The technique presented in [6] follows the SPE methodology very closely, generating the same kind of models as in [23], but it cannot take as input UML files produced with standard editors. In [16] UML models are transformed into Petri Nets, but the contention for hardware resources is not considered. In [5] it is presented a transformation from UML to Stochastic Process Algebra.

The performance research group from Carleton University has implemented a few UML-to-LQN transformations in different languages (such as graph-rewriting language Progres, text transformation language XSLT and general-purpose language Java) and was the first to use the standard UML metamodel libraries that were current at the time and the standard performance profiles SPT [17] and MARTE [18].

The most comprehensive model transformation of the Carleton group, which takes as input a number of different software models (including UML+SPT and UML+MARTE) and generates a number of target performance models (such as LQN, QN and Petri nets) is the PUMA transformation [26][27] and its extensions for Service-oriented Architecture, PUMA4SOA [1].

PUMA uses an intermediate model called Core Scenario Model (CSM) [27]. This way, PUMA succeeds in minimizing the large semantic gap between UML models and performance models and reduces the complexity of the transformation at the cost of having two separate transformations: one from UML+MARTE to CSM [21] [1] and another from CSM to LQN [22].

Comparing our light-weight Epsilon transformation with PUMA, our transformation goes directly from UML+MARTE to LQN, eliminating any intermediate model. Thus, the transformation is faster (as there is no need to generate and store an intermediate model) and supports easily inter-model traceability. Other differences stem from the languages used to implement the transformations. Our transformation is developed in Epsilon, a declarative/imperative language specialized for model transformations, which offers more powerful and concise language constructs; also, the Epsilon engine takes over a number of tasks (such as what rule to apply next) that must be handled explicitly by a Java transformation. On the other hand, PUMA was developed in Java, a general purpose language that does not provide built in operations to help in navigating the source model or connecting target model elements together, which makes the transformation longer and more complicated.

One of this paper's contributions is generating cross model trace links with the Epsilon Transformation Language (ETL), which supports generating trace links for each executed rule. As already mentioned, establishing trace links between source and target elements allows for tracking, analyzing and propagating the impact of change which results from evolving software models. There is a considerable difference in complexity between the traceability model in our transformation and that proposed in PUMA4SOA, an extension of PUMA for Service-Oriented Architecture [1]. Due to the use of the CSM intermediate model in PUMA4SOA, three traceability metamodels are necessary: UML-to-CSM, CSM-to-LQN and LQN-to-UML. This complicates not only the generation of the tracelinks, but also their navigation. In our case, we defined only one trace links metamodel, and each transformation execution generates one set of tracelinks from the software to the performance model, as shown in Figure 1.

The last objective of this paper handles the feedback of performance results to the original software model, giving the developers the opportunity to see the software model and its performance results in the same file. This could be used to complete the automation process in the performance improvement approach based on software antipatterns proposed in [8]. So far, the method for detecting antipatterns takes as input an XML file built by hand, which combines information about the software model and the performance results. This step could be automated now by applying our approach.

3. SOURCE AND TARGET MODELS

3.1 Source Model

The source model taken as input by the transformation is a UML 2.5 [19] software model annotated with MARTE [18] performance information. The source model contains two types of UML diagrams: a deployment diagram representing the structure of the system and one or more activity diagrams representing the behavior of selected key performance scenarios.

The deployment diagram contains a set of UML nodes stereotyped as <<*device*>> that represent physical computational resources with processing capability, and a set of artifacts representing software components, each deployed on a device. Each activity diagram represents a scenario that is the realization of a use case, and models the interaction between software components. The behavior of each participating component is modeled inside an *ActivityPartition* (also known as a swimlane) which belongs to an *ActivityGroup*. A swimlane contains different types of action nodes and control nodes linked together by edges. There are different types of action nodes, such as: a) *AcceptEventAction* - executed when an event has been triggered; b) *SendSignalAction* - responsible for creating and transmitting signal instances to the target object; c) *CallOperationAction* - transmits a message representing an operation call request to the target object and waits until a reply is received; and d) *Opaque*

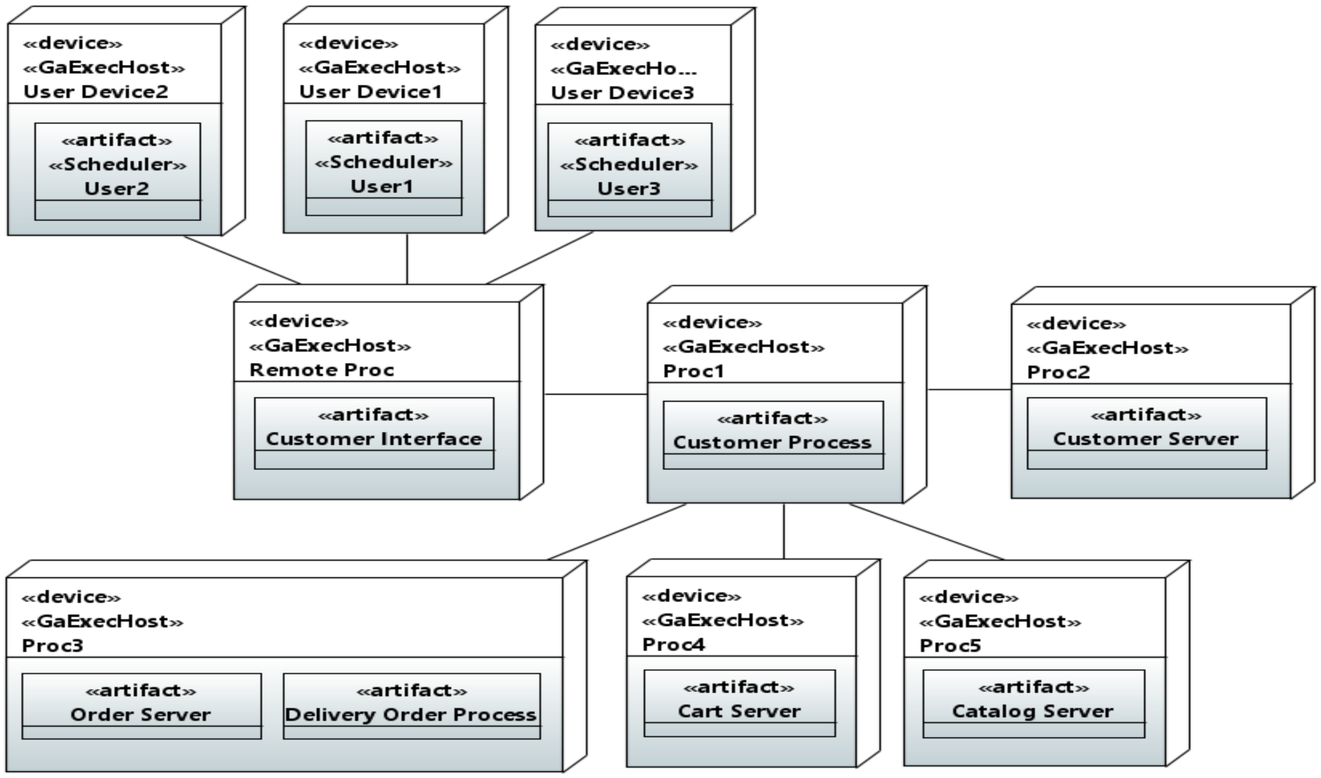

«device» «GaExecHost» User Device2
«artifact» «Scheduler» User2
«device» «GaExecHost» User Device1
«artifact» «Scheduler» User1
«device» «GaExecHo... User Device3
«artifact» «Scheduler» User3
«device» «GaExecHost» Remote Proc
«artifact» Customer Interface
«device» «GaExecHost» Proc1
«artifact» Customer Process
«device» «GaExecHost» Proc2
«artifact» Customer Server
«device» «GaExecHost» Proc3
«artifact» Order Server
«artifact» Delivery Order Process
«device» «GaExecHost» Proc4
«artifact» Cart Server
«device» «GaExecHost» Proc5
«artifact» Catalog Server

Figure 2. Deployment Diagram of E-commerce System

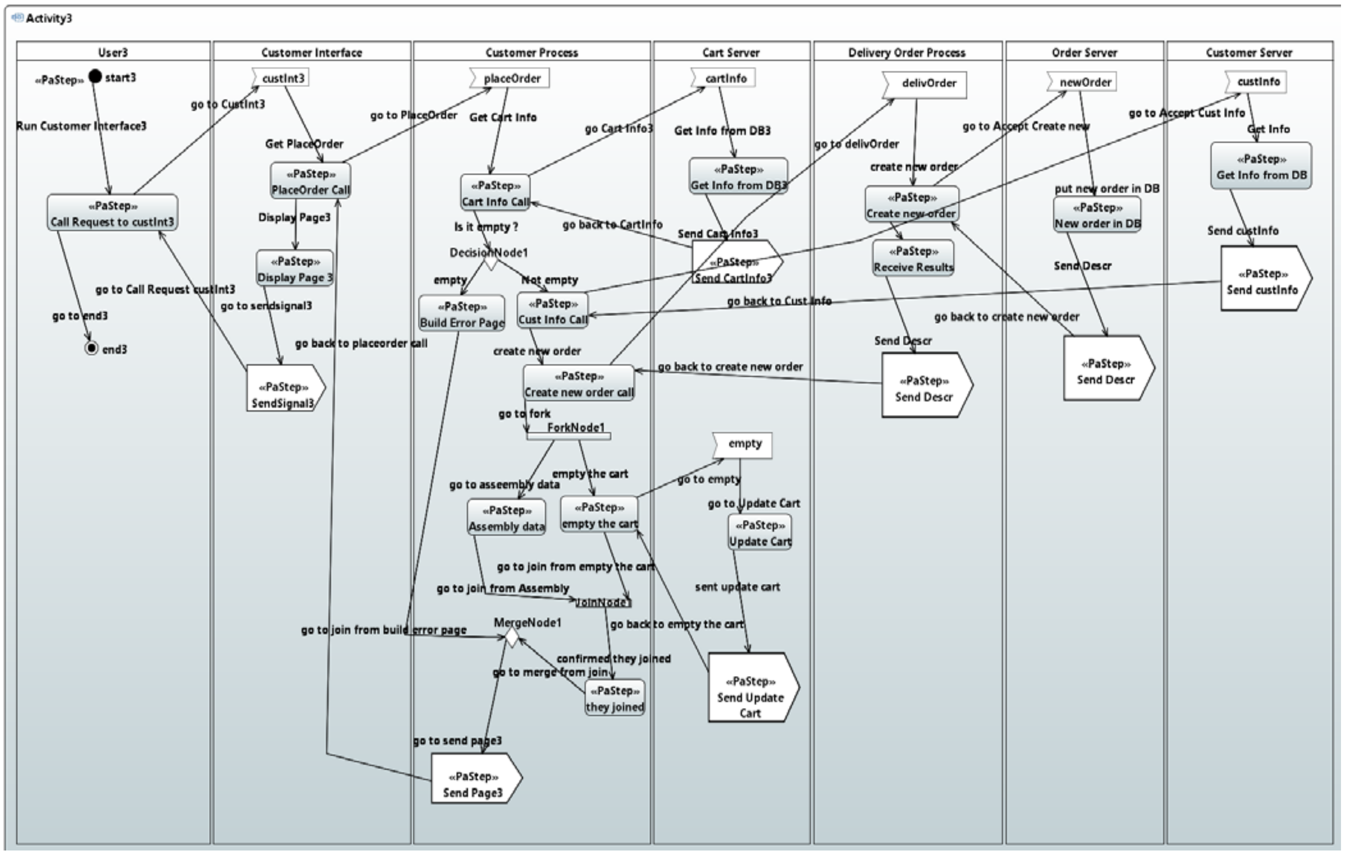


Figure 3. Activity Diagram of PlaceOrder Scenario

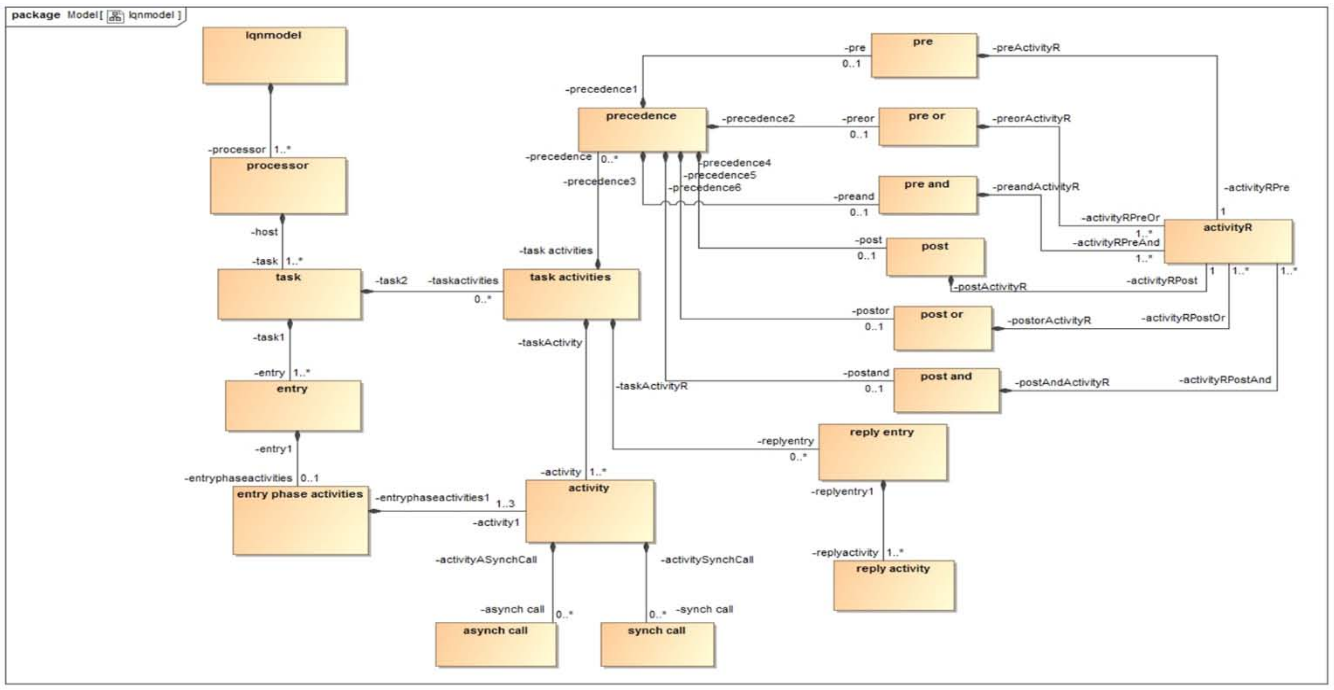


Figure 4. LQN metamodel designed for the Epsilon ETL transformation

Action - a type of UML abstract class considered as an executable node included within the behaviour. The control nodes are responsible for the flow of tokens between other nodes. Examples of control nodes are the initial node which indicates the starting point of the execution of the scenario and the final node which indicates the termination point of the execution. *ForkNode, JoinNode, MergeNode*, and *DecisionNode* are other examples of control nodes. Other type of model element is the *ControlFlow*, which is an activity edge responsible for passing tokens from its source node to its destination node. The activity edges interconnect activity nodes to form a graph that represents the behaviour of an activity as a sequence of subordinate units. In this paper we use the example of an e-commerce system model introduced in [7] as the source model for our transformation. The system contains three performance-critical use cases selected for performance analysis: *Browse Catalogue*, *Browse Cart*, and *Place Order*. Figure 2 represents the annotated deployment diagram of the system, showing the run-time architecture and the allocation of software components to hardware processing nodes. The system has three classes of customers with a population of *$N1*, *$N2* and *$N3* users, respectively. (Note that *$N1*, *$N2* and *$N3* are variables in the MARTE annotations). Each of the users is deployed on its own UserDevice host. In order to insure this, the multiplicity of *UserDevice1* is *$N1*, and so on.

Each class of users is executing repeatedly the use case corresponding to its class. The scenarios that represent the realization of the three use cases are modeled by three activity diagrams. The activity diagram for *PlaceOrder* scenario is given in Figure 3, while the other two can be found in [11].

In order to run the transformation successfully and get the expected results, the source model needs to satisfy some assumptions (the complete list can be found in [11]). Here are a few examples of such constraints. The namespace for each device element needs to be initialized to the UML element containing it; also the namespace for each artifact needs to be initialized to the device containing it. *ControlFlow* has a property called *inPartition*, which must be set only if the control flow is defined inside an *ActivityPartition*. For those *ControlFlow* representing call requests that cross the border between *ActivityPartitions*, the *inPartition* property does not have to be set.

3.2 Target Model

The target model for this transformation is the Layered Queuing Network (LQN) [12][13][14]. LQN is a performance model that is extended from queuing network and can represent nested services (i.e., a server may also be a client to other servers). A LQN model is an acyclic graph whose nodes are either software tasks (parallelograms) or hardware devices (circles) and the arcs denote service requests. Figure 5 shows the LQN model generated from the e-commerce example. Existing analytic LQN solvers compute the steady-state performance of a system with static allocation of resources. In the case where the resources are dynamically allocated on demand at run-time, the steady-state solution for each configuration of interests must be computed separately.

The tasks with outgoing but no incoming arcs play the role of clients (also called reference tasks), the intermediate nodes with both incoming and outgoing arcs are usually software servers and the leaf nodes are hardware servers. A software or hardware server node can be either a single-server or a multi-server. Software tasks have entries corresponding to different services (represented as smaller parallelograms inside the tasks).

The LQN metamodel is shown in Figure 4, and is based on the XML schema defined in the LQN user manual [14]. The Epsilon transformation engine, however, requires that the target metamodel be represented in EMF Ecore (the metamodeling language of the underlying platform Eclipse EMF [9]). The Eclipse framework offers a language called Emfatic, designed to represent EMF Ecore models in textual form. Therefore, we used the Emfatic language to express the metamodel from Figure 5 in a textual form, which in turn was converted into EMF Ecore. Like the XML-based metamodel from [14], the root model element of

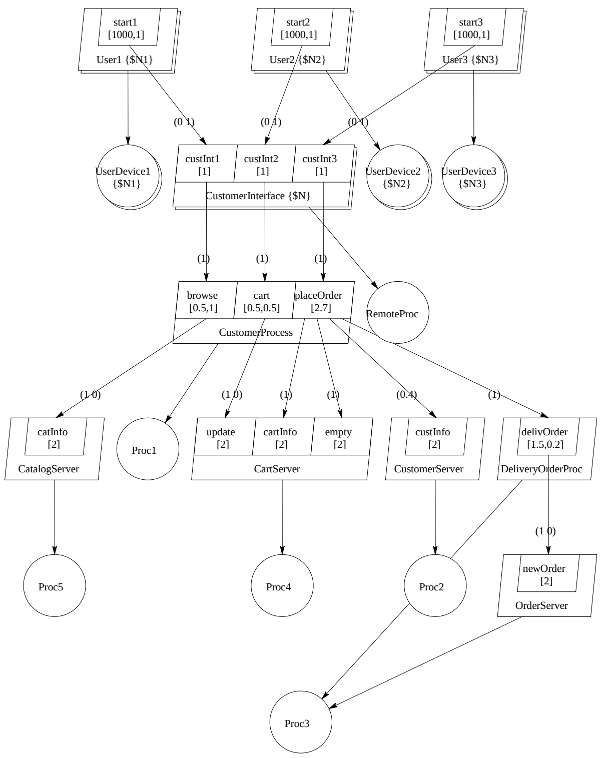


Figure 5. LQN model generated from the e-commerce software model from Figures 2 and 3

the LQN metamodel is lqnmodel, that is composed of one or more *processor* model elements by using composition associations. In other words, *lqnmodel* and *processor* have a whole-part relationship, following the hierarchy of the XML-based metamodel.

Processor is composed of *tasks*, which in turn is composed of *entries* or *task-activities*. Entry is the parent of *entry-phase-activities* model element, which is the container of *activity* model element. *Activity* is the parent of children of type *synch-call* and *asynch-call*. *Task-activities* element is composed of elements of three types: *activity*, *precedence* and *reply-entry*. *Reply-entry* is the parent of *reply-activity*. In addition, the elements named *pre*, *pre-or*, *pre-and*, *post*, *post-or* and *post-and* are all children of *precedence* model elements.

4. TRANSFORMATION

4.1 Transformation Process

As already mentioned, the transformation from UML+MARTE software model to LQN performance model is implemented in Epsilon, a family of languages, such as Epsilon Object Language (EOL) and Epsilon Transformation Language (ETL) for different model management tasks including model transformation, comparison, validation, etc. [15].

In order to generate LQN models from UML+MARTE software models we follow a multi-steps process described below:

a. *Building the source model*. The first step is building the UML software model with performance annotations as a source model for the transformation. An open source UML editor Papyrus [10] was used to build our source model. Papyrus development is supported by PolarSys, an Eclipse Industry Working Group created by large industry players and by tools providers to collaborate on the creation and support of Open Source tools for the development of embedded systems.

b. *Pre-transformation:* This is an optional step for checking and refining the source model in order to discover and eliminate bugs or fix missing data before feeding it to the next step, the main model transformation. An example is checking the *inPartition* attribute of *ControlFlow* elements (as mentioned in section 3.1). Setting *inPartition* attribute was automated by using Epsilon Object language (EOL)[15].

c. *Main Transformation:* developed in ETL language, it generates an initial LQN model in XML format that needs some minor extra processing to be in a format acceptable for the existing LQN solver tool.

d. *Post-Transformation:* the XML file for the initial LQN model has to be modified to be exactly conform to the XML schema [14]. The modification needs two steps. First, we change tag names by inserting dashes '-', which are not accepted by Emfatic, but are used in the XML schema [14]. Second, we add a solver-param element that cannot be derived from the source model. The modification for the tag names is done automatically by executing Java code but adding solver param was done manually.

The multi-steps transformation process was automated by using an orchestration workflow solution provided by Epsilon, extended from ANT [25]. Code Fragment 1 represents the template for the ANT file. Each workflow in the ANT file represents a project, each project has a number of targets, and each target has a number of tasks. The target can also depend on other target that has to be executed first. The default target is executed when the whole project is executed.

4.2 Mapping from Source to Target Model

In this section we present the mapping between UML+MARTE and LQN performance model, as shown in Table 1. Figure 6 illustrates at a high-level the mapping between source and target model elements, shown by red arrows. The elements in the UML are annotated with stereotypes from the MARTE profile (especially the performance analysis PAM subprofile) to bridge the gap between the UML and LQN performance models.

```
<?xml version="1.0"?>
<project default="main">
<target name="loadModels">

   <!--This part is to load the models(source,target) and their
metamodels-->

</target>
<target name="Execute" depends="loadModels">

   <!--This part is to load ETL file and EOL file -->

</target>
<target name="main" depends="Execute">
  <java classname="tagPackage.TagModifier" classpath="bin">

    <!--This part is to load a Java class with a method that sends
   the initial LQN model file as input argument and returns LQN
   model file after changing the tag names as output argument-->

   <arg value="initial LQN model file"/>
   <arg value="modified LQN model file"/>
  </java>
</target>
</project>
```

Code Fragment 1. ANT build file

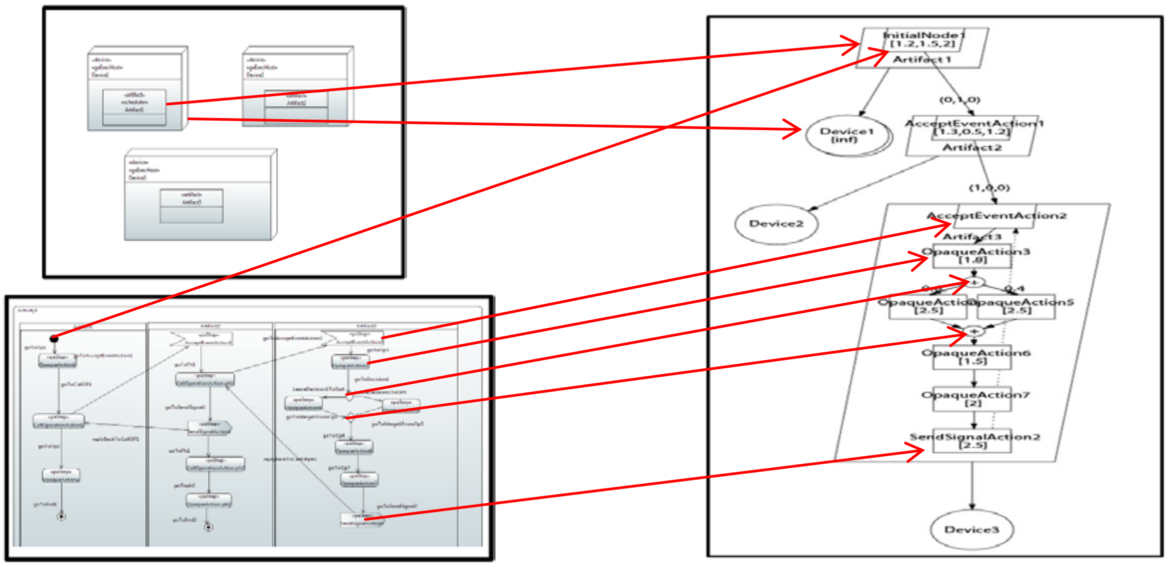


Figure 6. High-level view of the mapping between the source and target models

For instance, a *device* in the deployment diagram is stereotyped with*"GaExecHost"* to show execution resources [18] and is mapped to *processor* in the LQN model. An *artifact* is annotated with *"Scheduler"* that represents a kind of *ResourceBroker*, which creates access to its brokered *ProcessingResource* or resources following a certain scheduling policy [18]; an *artifact* is mapped to *task* in LQN.

Table 1. Mapping from UML+MARTE to LQN elemen

UM Model Element	MARTE Stereotype	LQN Element
Model	None	lqnmodel
Deployment Diagram		
Device	GaExecHost	processor
Artifact	Scheduler	task
Activity Diagram		
AcceptEventAction	PaStep	entry
InitialNode	None	entry
OpaqueAction	PaStep	activity
CallOperationAction	PaStep	activity
SendSignalAction	PaStep	activity
ControlFlow	None	precedence
DecisionNode	None	precedence
MergeNode	None	precedence
JoinNode	None	precedence
ForkNode	None	precedence
ControlFlow	None	synch-call
ControlFlow	None	asynch-call

In the activity diagram *AcceptActionElement*, *Initial Node*, *OpaqueAction*, *CallOperationAction* and *SendSignalAction* are annotated with *"PaStep"* stereotype, which is a type of *"GaStep"* and inherits its properties; "*PaStep*" can be applied to UML *action* or *message* to indicate behavior steps. *AcceptEventAction* and *InitialNode* are mapped to LQN *entry* element. *OpaqueAction, CallOperationAction* and *SendSignalAction* are mapped to LQN *activity* element. The rest of the elements in the activity diagram, such as *DecisionNode*, *MergeNode*, *JoinNode* and *ForkNode* are mapped to *precedence* in LQN. *ControlFlow* has some special conditions that need to be checked in the transformation rule for mapping it to the correct LQN element. For example, if the control flow does not cross its partition boundary and it is not an edge for any control node such as *MergeNode* or *JoinNode,* then it can be mapped to *precedence* LQN element, otherwise it has to be mapped to *synch-call* or *asynch-call*. See [11] for more details.

4.3 Transformation Rules

In this section we provide some examples of the transformation rules used to map the elements from the UML+MARTE source model to its corresponding element in the LQN target model. The transformation includes an ETL module that has 17 matched rules (non-abstract and non-lazy) and 26 EOL operations. We present just a few of the rules and operations; for details about other rules see [11]. In ETL the matched rules correspond to the declarative style and the lazy rules to the imperative style [15]. We avoided using any lazy rules as recommended in the Epsilon book [15] to avoid any deterioration in the performance of the transformation, as using lazy rules can take longer to run.

An operation is a function used to verify some conditions or return a value that can be called from any rule. The concrete syntax for the ETL rule has the *rule* keyword followed by the rule name and by (*to, transform*) keywords for defining the source and target parameter. The rule's body is specified as a sequence of EOL statements. An ETL rule also can have optional parts, such as *guard,* which contains an EOL expression that has to be satisfied in order to run the rule.

Our first example is *Model2Lqnmodel* rule (as shown in code fragment 2), which generates the target element *lqnmodel* from the source element *Model.* Once the *Lqn processor* is generated it will be added as a child to the *lqnmodel,* which is the root of the target model and has a containment relationship with *Lqn processor* element. *Lqn processor* is generated by another rule called *Device2Processor,* which transforms each device element from the source model to a *processor* in the target model and then adds the newly created *processor* element to the container of children of *lqnmodel*. For representing and keeping the containment relationship, ETL provides an built in operation called *"equivalent()"* that resolves a target elements which has been transformed by other rules [15]. In other words, a *processor* element is generated from a *device* source element and is added to the collection *lqn.proceesor*. As we can see in the code fragment 2, the *transform* part has two variables: "m" representing the source element *Model* of type UML and "lqn" representing the corresponding target element of type *lqnmodel.* Then the name attribute of the source element *"m"* is assigned to the name attribute of the target element *"lqn"*.

```
rule Model2Lqnmodel
    transform m : UML!Model
    to lqn : lqnmodel!lqnmodel{
        lqn.name= m.name;
    var devices : Collection=m.getDevices()
    for(d:Device in devices){
       lqn.processor.add(d.equivalent());}
}
```

Code Fragment 2. Transformation rule Model2Lqnmodel

```
operation Model getDevices():Collection{
   return Device.all.select(d|d.namespace=self
         and d.hasStereotype("GaExecHost")); }
```

Code Fragment 3. Operation getDevices

```
operation UML!Class
hasStereotype(name:String):Boolean {
    var c: Sequence;
    c=self.getAppliedStereotypes();
   for(s:Stereotype in c){
        if(s.name=name){
            return true;}
   }
  return false; }
```

Code Fragment 4. Operation hasStereotype

```
rule Device2Processor
   transform d : UML!Device
   to p : lqnmodel!processor {
   guard: d.hasStereotype("GaExecHost")
   p.name=d.name;
   var schedpolicy:String;
   schedpolicy=getStereotypeOfDevice(d);
   p.scheduling=checkPSchedulerType(schedpolicy);
   var col : Collection=d.getArtifact();
   for(a:Artifact in col){
        p.task.add(a.equivalent()); }
```

Code Fragment 5. Transformation rule Device2Processor

Code fragment 3 shows the operation *getDevices* () that is called from the rule *Model2Lqnmodel* in the context of the "Model" source element and returns a collection of all "device" instances after verifying that each element in the collection (represented by d in the code) has the correct stereotype. This is done by passing it as a parameter to the operation called *hasStereotype ()* (see code fragment 4) that returns "*True*" if the applied stereotype is equal to the passed parameter. Operation *getAppliedStereotypes()* is an ETL built in method called to get all stereotypes that have been applied to a UML class since this operation was called within the context of the UML class.

The second example is the *Device2Processor rule* (code fragment 5) that transferred each device element of the UML source model specifically the deployment diagram to processor element of the LQN target model. This rule has a guard that has to be satisfied in order to execute it. The guard part is responsible for checking if the device instance has "*GaExecHost*" stereotype. It guarantees that only the device instances annotated with "*GaExecHost*" are mapped to processors. The next part of the rule is to initialize the processor's attributes name and scheduling. The name of the processor is initialized with the name of the corresponding device in the deployment diagram. The scheduling attribute is initialized by calling the operation *GetStereotypeOfDevice()* and passing the device from the source as parameter to check if it has a *Scheduler* stereotype , then gets the value of *otherSchedPolicy* which is one of the *Scheduler's* attributes. As mentioned before, *Scheduler* stereotype is kind of ResourceBroker, which gives access to its brokered ProcessingResource or resources following a certain scheduling policy [18]. The returned value is assigned to a variable that is given the name *schedpolicy* and then is passed as a parameter to *checkPSchedulerType()* operation to set the value of *scheduling* attribute of the processor. The last part of the rule is collecting the artifact instances by calling *getArtifact()* operation in the context of a device, that returns all artifact instances whose namespace is initialized to the respective device. As the processor has a containment relationship with tasks, each task element is a target created from an artifact element by other transformation rule and added to the p.task reference of the processor.

It is important to emphasize the fact that transformation rules have been built such that they mirror the hierarchy of the LQN metamodel, which is based on containment relationships for connecting its elements together. The rules follow the same containment relationships to connect each generated target element with its parent/children by using the *equivalent ()* built in operation provided by ETL to resolve the source elements from their corresponding target elements transformed by other rules. That means, that, when applying *equivalent ()* to a source element, it invokes the applicable rule to generate the counterparts of the element in the target model [15].

For example, the *lqnmodel* root has a containment relationship with *lqn* processors elements, based on that all generated processors elements have to be added as a children to the *lqn* root target element by using *equivalent()* as explained before, to invoke *Device2Processor* rule that generates the processors elements. In the same way, *Device2Processor* rule added all generated tasks as children of the generated processor element in the target model, since each processor has a containment relationship with its tasks in the LQN metamodel. *Device2Processor* rule in turn invokes *Artifact2Task*, which invokes one of four rules to generate LQN entry type from either a UML *AcceptEventAction* or a UML *InitialNode.* It also invokes the corresponding rules for *OpaqueAction*, *CallOperationAction* and *SendSignalAction* to generate LQN elements of activity type. The rule which generates *CallOperationAction* invokes one of the two rules that transform a UML control flow to *LQN synchcall or LQN asynchcall. Artifact2Task* rule also invokes the rules that transfer *UML ControlFlow* and some of the *UML ControlNode,* such as *DecisionNode, MergeNode, JoinNode and ForkNode* to *LQN precedence.*

4.4 Traceability Model

In MDE traceability plays an important role for establishing links between source model elements and target model elements, in order to track, analyze and propagate the impact of changes which results from evolving the software models. In this section we present the steps we followed to generate cross-model trace links. Since each model has to conform to its metamodel, the trace model has to conform to its metamodel. The authors of [20] argue that case-specific traceability metamodels can be more specialized according to each traceability scenario and therefore the possibility of establishing illegitimate traceability links is reduced. This is unlike in general-purpose traceability metamodels that give the ability to create any number of trace links between elements from models regardless of their type, which gives a good chance for creating illegitimate traceability links. Our *Trace* metamodel similarly to its counterpart from [20], has a *Trace* class which has a containment relationship with *Tracelink* class. However, our *Trace* metamodel is different from [20] in that the *Tracelink* class has two attributes *sourceType* and *targetType* representing the source element type and the target element type. Another thing that is different, instead of having *sources* and *targets* in Tracelink class as references to source and target element instances, it has two attributes *sourceName* and *targetName*, representing the name of the source and target element instance, respectively. The way we used the *Trace* metamodel is discussed later in section 5.

Epsilon facilitates generating Trace model automatically when executing transformation modules, specifically the Post block. ETL module can have Pre/Post blocks, where the post block is executed in the order in which it has been specified after executing the transformation [15]. We create a new trace link for each rule executed by using *transTrace* that is a global variable set up automatically by ETL. Also *getSource()* and *getTargets()* are methods of the public class *Transformation* in *org.eclipse.epsilon.etl.trace Package* and are used to retrieve the source and target elements instances.

5. FEEDBACK OF PERFORMANCE RESULTS TO THE SOFTWARE MODEL

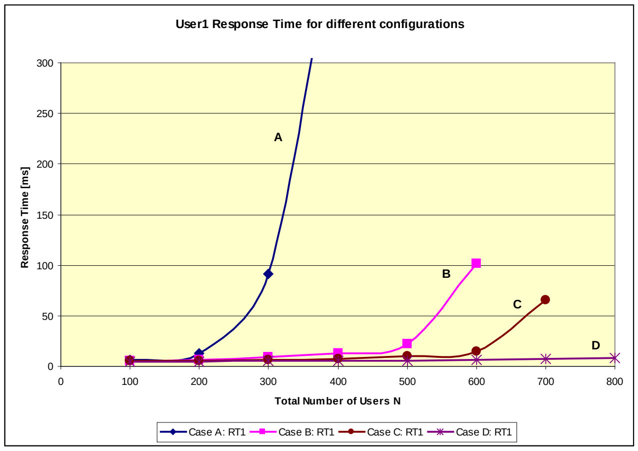
Our goal here is to feed back the performance results to UML software model during the round trip performance analysis. The developer can look at the results with the UML editor, analyze them, decide to make changes in the software model and repeat the process until satisfied with the overall performance.

In order to accomplish this goal, our approach in this round trip performance analysis depends basically on reading two files: first the *Trace Model* file that is generated automatically as a result of running the transformation, and secondly the XML file obtained from the LQN solver after solving the LQN model. Having the Trace model helps in matching each target element with its source element by following the trace link in the reverse way and then feedback the performance results to the corresponding source element as values of performance stereotype attributes. The source model is annotated with MARTE stereotypes that have performance attributes such as utilization, throughput and response time.

We implemented the proposed approach in EOL language that allows us to read two files: the Trace model and the XML file with the performance results and write that results to the third file, the original UML software model annotated with MARTE profile. EOL does not only provide a mechanism to read and write from/to files, but also facilitate to create, query and modify XML documents. The procedure starts by querying the XML file with the performance results, and then querying the Trace model and getting the *sourceName*, *sourceType* , *targetName and targetType* attributes for each trace link. Next the *targetName* attributes are used to get the corresponding source attribute name. It is worth stressing that matching is not done only by name, but by type as well. Once the source element has been identified, then we can retrieve its applied stereotype in the UML software model and set all its performance attributes to values corresponding to the LQN results. The purpose is to help the designers to better understand the performance results in terms of software model concepts rather than performance model concepts. The software developers may not be familiar with the performance model principles, but are certainly familiar with the software model. We applied our approach to the e-commerce application and fed back the performance results to the UML software model elements.

6. PERFORMANCE ANALYSIS

This section presents a brief performance analysis of the e-commerce system model introduced as an example in section 3, following the performance round-trip approach several times for different configurations corresponding to different resource multiplicities (both hardware and software). The software model is given in Figure 2 and 3, and the automatically generated LQN model in Figure 5. The response time and throughput performance results for artifact User1 are given in Figures 7.a and b.



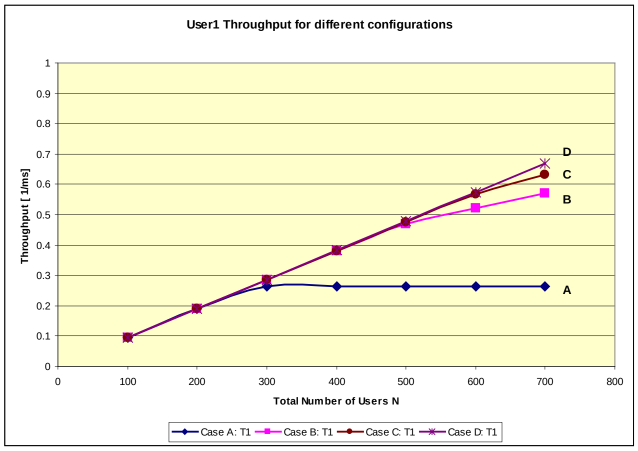


Figure 7.a) Response time and b) Throughput for class1 users

Please note that the e-commerce case study has been taken from literature [9] in order to minimize the author bias, which could be a potential threat to the validity of a proposed method when all test cases are created by the method's author.

As mentioned in section 3, there are three classes of users, each running a different scenario: *BrowseCatalog*, *BrowseCart* and *PlaceOrder*. The think time for each user is 1s. Class1 has the largest number of users. In our experiments, we kept a constant ratio between the numbers of users: for 48 users in class1, there is one user in class2 and one user in class3. For each configuration, we solved the model under a variable load (from 100 to 800 users in total). Figure 7.a shows the average response time for a class1 user for four configurations described below, while Figure 7.b shows the average throughput.

A. This is the base case, where the multiplicities of all tasks and processors are 1, with the exception of the *User* tasks (each with its own processor) and the *CustomerInterface* task, which has a thread for every *User*. The results show that the response time and throughput curves have each a knee starting at around 200 users; the performance deteriorates very quickly after the knee, so the operating point should be selected before the knee. The analysis of task and processor utilizations shows that this is a typical case of software bottleneck, with *CustomerProcess* being the task that saturates first, limiting the concurrency level and the utilization of resources below the bottleneck. An appropriate solution is to raise the number of threads for *CustomerProcess* – which is purely a software solution.

B. We raised the number of threads of *CustomerProcess* from 1 to 50. The results show that the response time and throughput improve considerably and the knee of the curves moves to the right, being able to accommodate more than double the number of simultaneous users before the knee than in Case A. The next bottleneck is *CatalogServer* processor, followed immediately by the *CatalogServer* process. The solution is to raise the number of software threads of *Catalog Server* to 50 its processor multiplicity to 5.

C. We applied the solution described above. The performance improvement from case B to C is less important than from A to B, but still, the knee moves further to the right, accommodating at least 100 more simultaneous tasks. The next bottleneck is the processor of *CustomerProcess*.

D. After increasing the multiplicity of the *CustomerProcess* processor to 5, the response time improves further and the knee moves beyond 800 users (where we stopped increasing the workload).

This simple example illustrates how the LQN performance model can be used to predict the performance effect of different configuration changes. Here the actual diagnosis of performance problems (i.e., detecting the system bottleneck) and finding a solution for alleviating the problem was done by a human analyst. In the future, we intend to integrate our performance round-trip approach with performance diagnosis algorithms that will enhanced the level of support offered to software developers by MDE tools.

7. CONCLUSIONS

The contribution of this paper is a model to model transformation to generate LQN performance models from UML+MARTE software models. The transformation was developed with Epsilon, a specialized language family that helped us to develop a more compact transformation than with a general purpose language (such as Java), easier to understand and maintain. Another contribution is the generation of cross-model trace links and their use to import performance results into the initial software model. The advantage of the one-step transformation is the ability of generating automatically along the transformation direct trace links between the source and target model. After solving the performance model with an existing solver, the performance results (such as response time, throughput and utilization of different model elements) are fed back to the software model by following the cross-model trace links. The software developers have access to the performance results as MARTE stereotype attributes, using a standard UML editor.

The transformation presented in this paper has a number of limitations. One limitation is that the source model must be built according to the description from section 3.1: a deployment diagram which shows the allocation of software components to hardware processors, and one or more activity diagrams which models the behavior of key scenarios selected for performance analysis. Some developers may prefer to represent such scenarios by using sequence diagrams, so a future extension of the transformation will allow for the mapping of sequence diagrams to LQN model elements. Another limitation is that the transformation works as expected only if all the MARTE stereotypes and their attributes are correctly applied by the developers who built the UML source model. If stereotypes or their attributes are missing where they are expected to be defined, the transformation crashes. We plan on making the whole process more user-friendly by defining pre-transformation operations that check the source model and either notify the developer if elements with undefined stereotypes and attributes are found, or assign default values to undefined attributes (for instance, missing CPU processing demands of <<*PaStep*>> can be assigned to zero). Such pre-transformation verifications of the source model would minimize the transformation errors due to incorrect input models.

Another direction for future work is to integrate our approach with the approach presented in [8] to automatically detect and fix performance problems based on performance antipatterns. Their approach is not fully automated, as the software model and performance results exits in different models and are brought in the same file manually. Another future work direction is to apply our approach for generating cross-model trace links to other analysis models for other nonfunctional requirements (such as reliability, availability, safety).

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Discovery Grant program.

REFERENCES

[1] Alhaj, M., "*Automatic Derivation of performance Models in the Context of Model-Driven SOA*", Ph.D. Thesis, Department of Systems and Computer Engineering, Carleton University, January 2014.

[2] Arcelli, D., Cortellessa V., 2013. "Software Model Refactoring Based on Performance Analysis: Better Working on Software or Performance Side?" *In Proceedings FESCA* 2013, Pages 33–47.

[3] Arcelli, D. and Cortellessa, V. 2015. "Assisting Software Designers to Identify and Solve Performance Problems". In *Proceedings of the 1st International Workshop on Future of*

Software Architecture Design Assistants - FoSADA '15. (2015), 1–6.

[4] Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M. "Model-based performance prediction in software development: a survey", *IEEE Transactions on Software Engineering*, vol. 30, N.5, pp.295-310, 2004.

[5] Cavenet, C., Gilmore, S., 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 (WOSP 2004)*, pp. 74-83, Redwood City, CA, Jan 2004.

[6] Cortelessa, V., Mirandola,R., "Deriving a Queueing Network based Performance Model from UML Diagrams*", in Proc. of 2nd ACM Workshop on Software and Performance* (WOSP'20004), pp.58-70, Ottawa, Canada, 2000.

[7] Cortellessa V., Di Marco, A., Inverardi, P., Model-based Software Performance Analysis, Springer, 2011.

[8] Cortellessa, V., Di Marco, A., Trubiani,C.. 2014. "An Approach for Modeling and Detecting Software Performance Antipatterns Based on First-Order Logics." 391–432.

[9] Eclipse Foundation, "Eclipse Modeling Framework (EMF)", www.eclipse.org/modeling/emf/, last visited Jan 2, 2016.

[10] Eclipse Foundation, "Papyrus Modeling Environment", eclipse.org/papyrus, last accessed May 2016.

[11] Hassanzadeh Zargari, M.. 2016. "Automatic Derivation of LQN Performance Models from UML Software Models Using Epsilon, Master Thesis, Department of Systems and Computer Engineering, Carleton University, January 2016.

[12] Franks, R.G. "*Performance Analysis of Distributed Server Systems*", PhD Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.

[13] Franks, R.G., Al-Omari, T., Woodside C.M., Das,,O. and Derisavi,, S. "Enhanced modeling and solution of layered queueing networks", IEEE Transactions on Software Engineering, 35(2):148–161, 2009.

[14] Franks, R.G., Maly, P., Woodside, C.M., Petriu, D.C., Hubbard, A., Mroz, M.,"Layered Queueing Network Solver and Simulator User Manual", Department of Systems and Computer Engineering, Carleton University, 2015.

[15] D. Kolovos, L. Rose, A. García-Domínguez, R. Paige, The Epsilon Book, www.eclipse.org/epsilon/doc/book/, last updated July 2015.

[16] Lopez-Grao, J.P., Merseguer, J., Campos, J., "From UML Activity Diagrams to Stochastic Petri Nets: Application To Software Performance Engineering," 4th Int. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, (2004), pp. 25-36.

[17] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification", version 1.1, formal/05-01-02, January 2005.

[18] Object Management Group, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, formal/2011-06-02, 2011.

[19] Object Management Group, "OMG Unified Modeling Language" Version 2.5, formal-15-03-01, 2015.

[20] Paige, Richard F. et al. 2011. "Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering." Software and Systems Modeling 10(4):469–87.

[21] Petriu,D.B., Woodside, C.M., "An intermediate metamodel with scenarios and resources for generating performance models from UML designs", Software and Systems Modeling, Volume 6, Nb. 2, pp.163-184, 2007.

[22] Petriu,D.B., "*CSM2LQN – Transformations for the Generation of Performance Models from Software Designs*", Ph.D. Thesis, Department of Systems and Computer Engineering, Carleton University, 2014.

[23] Smith, C.U., *Performance Engineering of Software Systems*, Reading Mass., Addison Wesley, 1990.

[24] Smith, C. U., Williams, L.G., *Performance Solutions*, Addison-Wesley, 2002.

[25] The Apache Ant Project. http://ant.apache.org.

[26] Woodside, C.M., Petriu, D.C., Petriu,D.B, Shen, H. Israr, T. Merseguer, J. "Performance by Unified Model Analysis (PUMA)", Proc. of 5th ACM Workshop on Software and Performance, pp.1-12, Palma,Spain, July 2005.

[27] Woodside, C.M., Petriu, D.C., Merseguer, J., Petriu, D.B., Alhaj, M. "Transformation challenges: from software models to performance models", Software and Systems Modeling, Vol. 13, No. 4 (2014), Page 1529-1552