User-Friendly Approach for Handling Performance Parameters during Predictive Software Performance Engineering

Rasha Tawhid
School of Computer Science
Carleton University, Ottawa, ON, Canada
e-mail: rtawhid@connect.carleton.ca

Dorina Petriu
Dept. of Systems and Computer Engineering
Carleton University, Ottawa, ON, Canada
e-mail: petriu@sce.carleton.ca

ABSTRACT
A Software Product Line (SPL) is a set of similar software systems that share a common set of features. Instead of building each product from scratch, SPL development takes advantage of the reusability of the core assets shared among the SPL members. In this work, we integrate performance analysis in the early phases of SPL development process, applying the same reusability concept to the performance annotations. Instead of annotating from scratch the UML model of every derived product, we propose to annotate the SPL model once with generic performance annotations. After deriving the model of a product from the family model by an automatic transformation, the generic performance annotations need to be bound to concrete product-specific values provided by the developer. Dealing manually with a large number of performance annotations, by asking the developer to inspect every diagram in the generated model and to extract these annotations is an error-prone process. In this paper we propose to automate the collection of all generic parameters from the product model and to present them to the developer in a user-friendly format (e.g., a spreadsheet per diagram, indicating each generic parameter together with guiding information that helps the user in providing concrete binding values). There are two kinds of generic parametric annotations handled by our approach: product-specific (corresponding to the set of features selected for the product) and platform-specific (such as device choices, network connections, middleware, and runtime environment). The following model transformations for (a) generating a product model with generic annotations from the SPL model, (b) building the spreadsheet with generic parameters and guiding information, and (c) performing the actual binding are all realized in the Atlas Transformation Language (ATL).

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: *modeling techniques, performance attributes*. D.2.4 [**Software/Program Verification**]: *model checking*

General Terms
Performance, Design

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA.

Keywords
Model-driven development, performance model, Performance Completion, ATL, MARTE, SPL, UML.

1. INTRODUCTION
A Software Product Line (SPL) is a set of similar software systems built from a shared set of assets, which are realizing a common set of 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 [9].

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. In previous research [22][23][24] the authors of the paper proposed 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. Our starting point was 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 [18]. Such parameters appear as variables and expression in the MARTE stereotype attributes.

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 three big steps: a) instantiating a product model with generic performance parameters from the SPL model; b) binding the generic parameters to concrete values provided by the user and c) generating a performance model for the product from the model obtained in the previous step. The model transformation (a) was developed in our previous work [22][23]; step (b) represents the contribution of this paper; and the PUMA transformation (c) for generating performance models from annotated UML models has been developed previously in our research group [27].

Since step (b) requires input from the user, it is implemented by two transformations, as shown in Fig. 1: the first collects all the generic parameters that need to be bound to concrete values from the automatically generated product model and presents them to the developer in a user-friendly spreadsheet format, while the

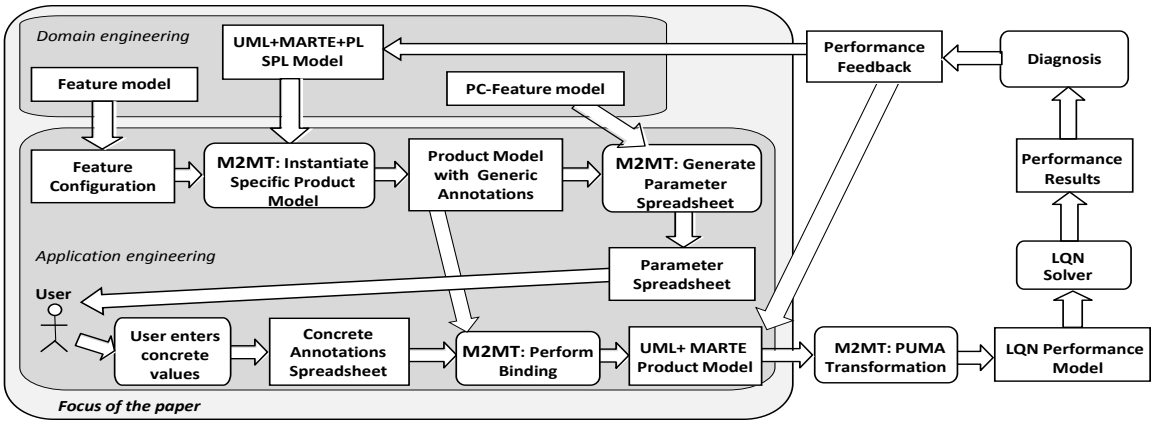

Figure 1. Approach for deriving a product performance model.

second performs the actual binding to concrete values provided by the developer. The list of annotation parameters presented to the developer contains not only the parameter name and the model elements it belongs to, but also some guiding information to help the user in providing concrete values, as explained in more detail in section 3.2.

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. proposed the concept of performance completions to close the gap between abstract design models and external platform factors [26]. Performance completions provide a means to extend the modeling constructs of a system by including the influence of the underlying platforms and execution environments in performance evaluation models. 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 so-called Performance Completion feature (PC-feature) model similar to [13]. 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 [23]. Therefore, our approach uses two feature models for a SPL: 1) a regular feature model for expressing the variability between member products (as described in Section 2.1), and 2) a PC-feature model introduced for performance analysis reasons to capture platform-specific variability (as described in Section 2.3).

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, as described in Section 3.2.

Dealing manually with a large number of performance parameters and with their mapping, by asking the developer 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. This paper proposes a model transformation approach 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.

We claim that the proposed technique for handling annotation parameters is user-friendly after comparing it with another approach used in earlier phases of our research, where the binding information was given as a set of couples {<generic_parameter>, <concrete_value>} created manually by the developer, after careful inspection of the generated UML model to extract all the parameters. The older approach required a lot of work from the developer and was error prone. The parameter file produced by hand contained no context information and no guidelines.

The proposed technique is illustrated with an e-commerce case study. The e-commerce SPL can generate a distributed application that can handle either business-to-business (B2B) or business-to-consumer (B2C) systems.

The paper is organized as follows: section 2 presents the domain engineering process where the SPL model, the regular feature model and the PC-feature model are constructed; section 3 presents the model transformations for generating a product model with concrete performance specifications; related work is discussed in section 4; and section 5 presents the conclusions.

2. 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 SPL assets created by the domain engineering process of interest for our research are represented by a multi-view UML design model of the family, called the SPL model, which

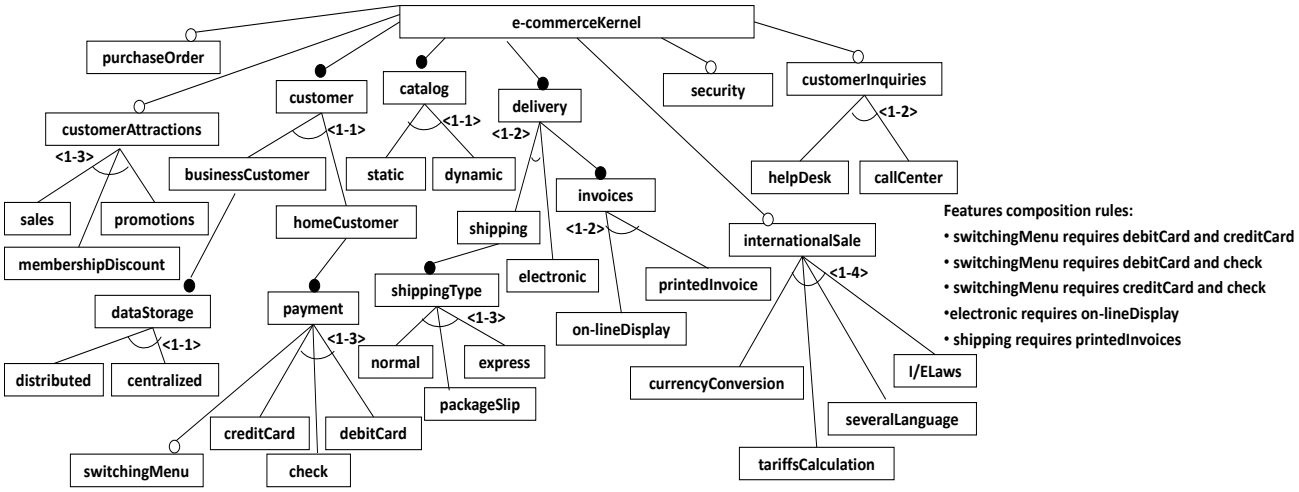

Figure 2. Feature model of the e-commerce SPL.

represents a superimposition of all variant products [22][23]. The creation of the SPL model employs two separate profiles: a product line profile introduced by Gomaa [12] 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, we create a PC-feature model to represent the variability space of performance completions.

2.1 Feature Model

The feature models are used in our approach to represent two different variability spaces. This section describes the regular feature model representing functional variabilities between products. An example of feature model of an e-commerce SPL is represented in Fig. 2 in the extended FODA notation, Cardinality-Based Feature Model (CBFM) [11]. 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*. 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 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.

2.2 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.

Note that 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.

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»*; such use cases are also 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. The structural view of the SPL is presented as a class diagram and variabilities are modeled in the same manner as the use case diagram (i.e., stereotyped as *kernel*, *optional* or *alternative*). For more details about the SPL use case and class diagrams see [22][24].

The behavioural SPL view is modeled as sequence diagrams for each scenario of each use case of interest. Fig. 3 illustrates the kernel scenario *BrowseCatalog*. *«GaAnalysisContext»* is a MARTE stereotype indicating that the entire interaction diagram is to be considered for performance analysis. Its tag indicates a list of annotation variables representing context analysis parameters:

{contextParams = $N1, $Z1, $ReqT, $FSize, $Blocks}

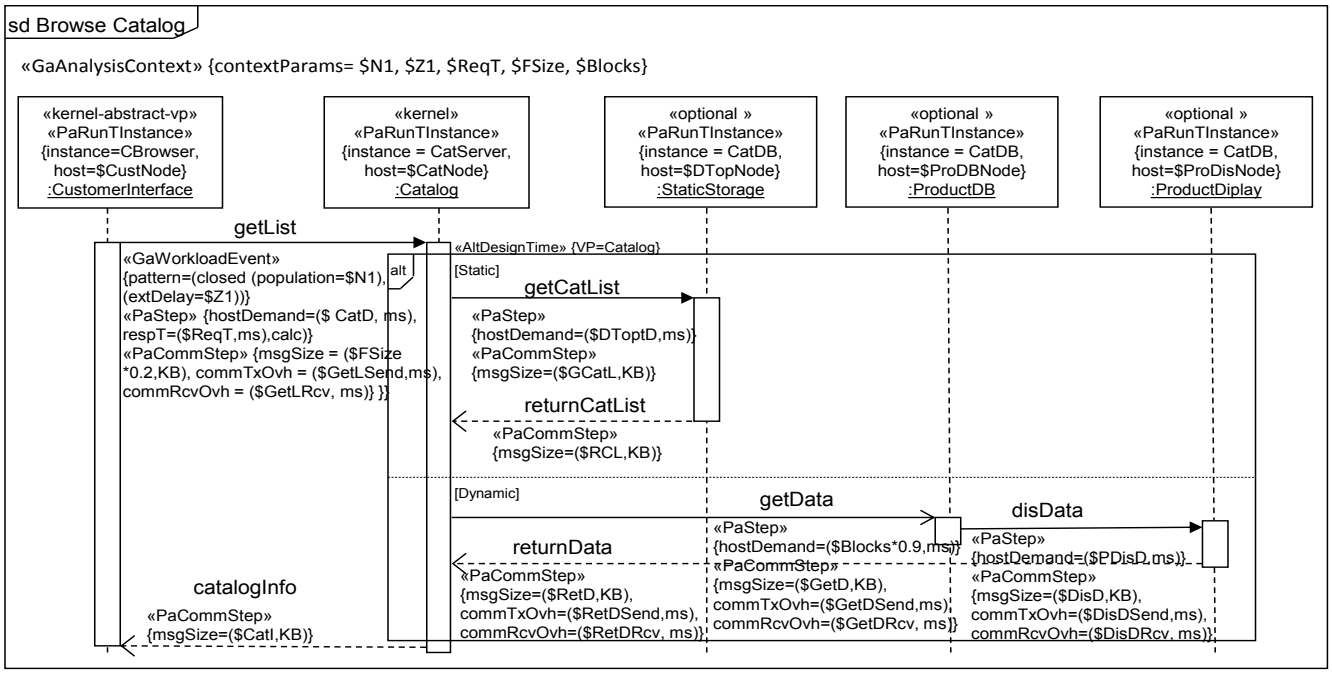

Figure 3. SPL Scenario Browse Catalog.

By convention, we use names starting with '$' for all MARTE variables to distinguish them from other identifiers and names.

The workload of a scenario is defined as a stream of events driving the system; a workload may be open or closed.

«GaWorkloadEvent» {pattern=$Patt}

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.

Each lifeline whose role is an active object is stereotyped as *«PaRunTInstance»*, providing an explicit link at the annotation level between a role in a behavior definition (a lifeline) and the runtime instance of a process or thread (active object). For example, the tag *{instance=CBrowser}* indicates the name of runtime instance of a process executing the lifeline role, while the tag *{host=$CustNode}* indicates the physical node from the deployment diagram on which the instance is running. In this case, the *host* is given by the generic parameter *$CustNode*, which will be bound later to a concrete host name.

Conceptually, a scenario represented by a UML sequence diagram is composed of units of execution named steps. MARTE defines two kinds of steps for performance analysis: execution step (stereotyped *«PaStep»*) and communication step (stereotyped *«PaCommStep»*). *«PaStep»* may be applied to an Execution Occurrence (represented as a thin rectangle on the lifeline) or to the message that triggers it. For instance, in Fig. 3, the message *getList* is stereotyped as an execution step:

«PaStep» {hostDemand = ($CatD, ms),
respT = (($ReqT, ms), calc)}

where *hostDemand* indicates the execution time required by the step, given by the variable *$CatD* in time units of milliseconds. The attribute *respT* corresponds to the response time of the scenario starting with this step; the variable *$ReqT* will save the calculated response time in milliseconds. The same message *getList* is also stereotyped as a communication step:

«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).

Since performance analysis depends on the software to hardware allocation, another structure diagram, which is not usually represented in the SPL models, has to be created in our approach.

2.3 SPL Performance Completion

In SPL, different members may vary from each other in terms of their functional requirements, quality attributes, platform choices,

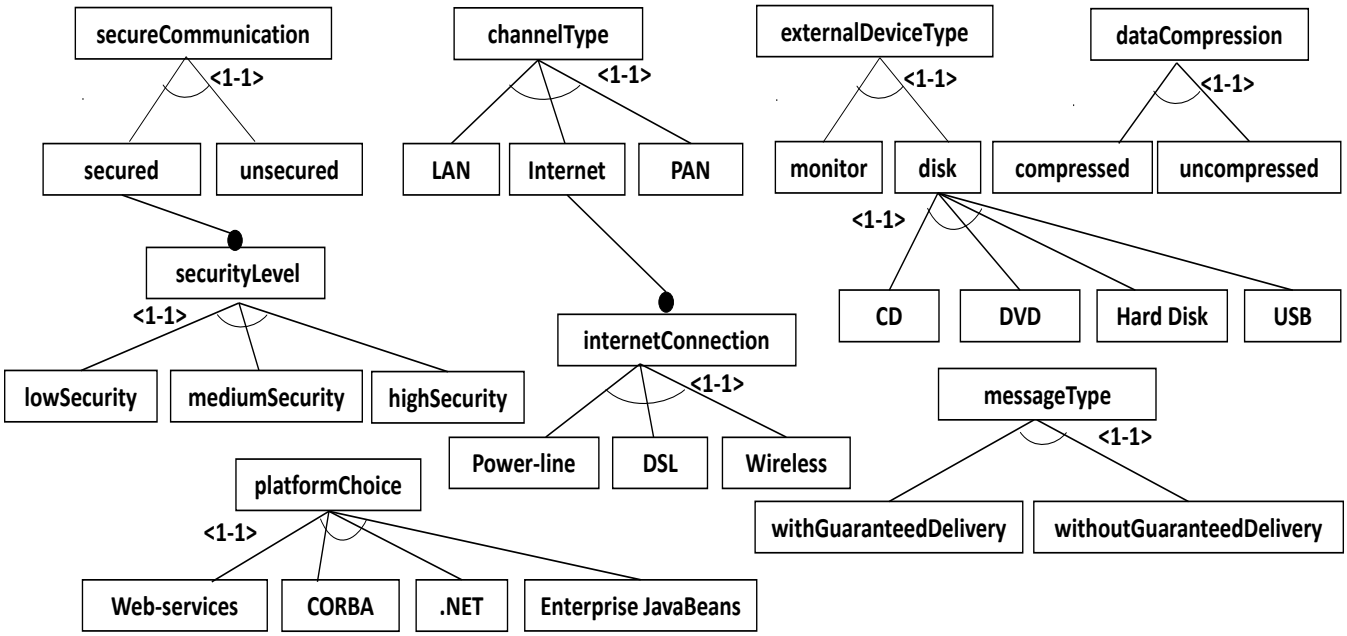

Figure 4. Part of the Performance-Completion feature model of the e-commerce SPL.

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 [26], 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.

This subsection covers the variability space of the performance completions and represents it through a Performance Completion feature model (PC-feature model) similar to [13]. Each feature from the PC-feature model shown in Fig. 4 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. Table 1 illustrates this type of mapping between PC-features and the design model, set up through the MARTE stereotypes attached to model elements.

Fig. 4 illustrates a part of the PC-feature model for our case study. Adding security solutions requires more resources and longer execution times, which in turn has a significant impact on system performance. We introduce a PC-feature group called *secureCommunication* that contains two alternative features *secured* and *unsecured*. The *secured* feature offers three security level alternatives depending on the size of the key used in the handshake phase and on the strength of the encryption and message digest algorithms used in the data transfer phase, as proposed in [15]. Each security level requires 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 type of physical communication channel stereotyped *«GaCommHost»* has different capacity for the amount of information that can be transmitted over it. As the channel's capacity increases, the latency time for transmitting data over this

Table 1. Mapping of PC-features to affected performance attributes

PC-feature	Affected Performance Attribute	MARTE Stereotype	MARTE Attribute
secureCommunication	Comm. overhead	PaCommStep	commRcvOvh commTxOvh
channelType	Channel Capacity Channel Latency	GaCommHost	capacity blockT
dataCompression	Message size Comm. overhead	PaCommStep	msgSize commRcvOvh commTxOvh
externalDeviceType	Service Time	PaStep	extOpDemand
messageType	Comm. overhead	PaCommStep	commTxOvh

channel decreases. Our example provides three different communication channels with three alternative connections for the Internet. The capacity and latency for each physical channel type are mapped to the attributes *capacity* and *blockT* of the stereotype *«GaCommHost»*.

Data compression requires extra operations that increase the processing time, but at the same time compression helps reducing the use of resources, such as hard disk space or communication channel bandwidth. Data compression/decompression is adding an overhead when sending and receiving a message, which is mapped to the attributes *commTxOvh* and *commRcvOvh,* respectively. However, compression also reduces the amount of data to be transferred and thus decreases the delivery time (e.g., a compression algorithm may reduce the size of data to 60% [13]).

Thus, the amount of compressed data transmitted over a physical channel is mapped to the attribute *msgSize*. Another communication mechanism that affects the delivery time of a message is whether the communication is with or without guaranteed delivery [13]; the effect is mapped to the *commTxOverhead* attribute.

The PC-feature group *platformChoice* includes different types of middleware such as CORBA, Web-services, etc., which will affect also the communication overheads. We may either map their effect to the *commTxOvh* and *commRcvOvh* attributes, or may use MARTE external operations described below.

MARTE provides specifically the concept of "external operation calls" to represent resource operations that are not explicitly modeled within the UML design model, but may have an impact on performance. The stereotype *«PaStep»* has two attributes: a) *extOpDemands*, an ordered set of identifiers for operations by external services which are demanded by this Step, in a form understood by the performance environment, and b) *extOpCount*, an ordered set of number of requests made for each external operation during one execution of the Step, listed in the same order as the demands. Examples of such external calls are middleware operations or disk operations hidden in database calls. Different types of external devices are represented in the PC-feature model by the feature *externalDeviceType*. Each device offers different operations times with different execution times. The invocation of such operations is represented by the attributes *extOpDemands* and *extOpCount* of an execution step.

It is important to note that the MARTE annotation contain both performance-affecting attributes of the product we want to analyze, as well as environment/platform characteristics. For instance, the CPU execution times of different scenario steps are indicated by the attributes *hostDemand* of *«PaStep»*. The size of a message from a sequence diagram represented by the attribute *msgSize* is a property of the software product, which may be modified by properties of the communication channel, such as compression/decompression. The product model obtained by the transformation presented in the next section includes both the performance attribute contained directly in the design model and the platform/environment factors corresponding to PC-features.

In order to automate the process of generating a user-friendly representation of the generic MARTE parameters that need to be bound to concrete values, the mapping between the PC-features and the performance attributes they affect needs to be specified, as shown in the next section. Also, each stereotyped class representing a feature in the PC-feature model has a tagged value indicating the list of the attributes it affects. For instance, the PC-feature *DataCompression* affects the attribute list {*msgSize*, *commTxOvh*, *commRcvOvh*}.

3. 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 model 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. We have implemented these model transformations in the Atlas Transformation Language (ATL). 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).

3.1 Product Model Derivation

This subsection describes briefly the first model transformation for generating a product model with generic performance annotations from the SPL model, which was developed by the authors in previous work [22][23][24].

Our derivation approach uses the mapping technique previously proposed in [24] to set up the mapping between a functional feature from the feature model and the model element(s) realizing the feature in the SPL model (both in the structural and behavioural views).

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 second step in the derivation process is to select the use cases realizing the chosen features. The product class diagram is derived in the third step in a similar way to the use case diagram. 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 [24].

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.

3.2 Generating User-Friendly Representation

The generic parameters of a product model 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. For example, Fig. 5 shows a part of the deployment diagram to be used for the scenario *BrowseCatalog*. The user has to decide for a product what is the actual hardware configuration 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. For instance, the user may decide to allocate the role *ProductDisplay* to the actual processing node *CatalogNode*.

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. For instance, a PC-feature configuration for a given message could be {*MediumSecurity*, *Compressed*, *CORBA*, *withoutGuaranteedDelivery*}.

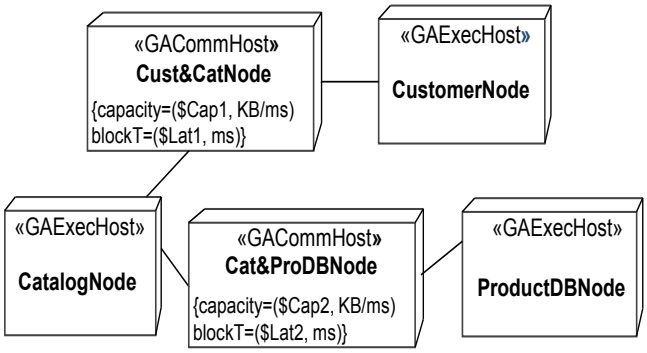

Figure 5. Part of a product deployment diagram.

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 *securedCommunication* affects only certain communication channels in a product 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 in the product model, as described in section 2.3.

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.

An example of such guiding information is the different overheads for sending and receiving secure messages. For instance, in [15] overhead data is provided for three security levels (low, medium and high): the handshake overhead is (10.2ms, 23.8 ms, 48.0 ms), and the data transfer overhead per KB of data is (0.104 ms, 0.268 ms, 0.609 ms). For instance, we used this data as guiding information in Fig. 6. For instance, the overhead for sending a message with low security level is (5.1+0.052*msgsize) and for receiving is (5.1+0.052*msgsize). For a given SPL, the performance analyst may tailor the guiding information to the platform and environment intended for performance analysis.

The process of generating the spreadsheets takes place after a specific product model is derived from the SPL model. Due to the large semantic gap between the source and target models of the transformation, we follow the example ”Microsoft Office Excel Extractor” [7] from the Eclipse/ATL website, which applies the principle of separation of concerns and breaks the transformation into a series of simpler transformations. This transformation series enables us to get some control over the order in which to navigate the ATL source models. The process is composed of four different model transformations:

a) From a specific UML product model into a Table model that contains several tables, one for each sequence diagram; each parametric performance annotation is represented as a table row;

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» secureCommunication	unsecured	No effect		
					secured			
				«exactly-one-of feature» securityLevel	lowSecurity	add (5.1+0.052*msgsize)		
					mediumSecurity	add (11.9+0.134*msgsize)		
					highSecurity	add (24.0+0.305*msgsize)		
				«exactly-one-of feature» dataCompression	compressed	increase by 2% ...5%		
					uncompressed	No effect		
							$GetLSend (ms)	
			commRcvOverhead	«exactly-one-of feature» secureCommunication	unsecured	No effect		
					secured			
				«exactly-one-of feature» securityLevel	lowSecurity	add (5.1+0.052*msgsize)		
					mediumSecurity	add (11.9+0.134*msgsize)		
					highSecurity	add (24.0+0.305*msgsize)		
				«exactly-one-of feature» dataCompression	compressed	increase by 2% ...5%		
					uncompressed	No effect		
							$GetLRcv (ms)	

Figure 6. Part of the generated Spreadsheet for the scenario Browse Catalog.

b) From the Table model into a SpreadsheetMLSimplified model that represents (as the name says) a simplified subset of the spreadsheetML XML used by Microsoft to import/export Excel workbook's data from/to XML;

c) From the SpreadsheetMLSimplified into an XML model;

d) The XML model created in the previous step is re-written as an XML file which can be directly opened by Microsoft Excel.

The mapping between PC-features and the corresponding performance attributes takes place during the transformation (a). Each MARTE attribute gets the name of the PC-features that have an impact on this attribute attached to it. For instance, the attribute *msgSize* is associated with the PC-feature *Data Compressed*. Another association is between the MARTE attribute *host* annotating a model element of type lifeline and the list of all available deployment nodes from the deployment diagram. After the user selects a PC-feature combination for each model element, he/she can delete the remaining unselected PC-features from the spreadsheet, ending up with a small set of rows containing annotations that need to be bound to concrete 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.

The four transformations are implemented in the Atlas Transformation Language (ATL) [1]. An ATL transformation is composed of a set of rules and helpers. The rules define the mapping between the source and target model, while the helpers are methods that can be called from different points in the ATL transformation. The rules of the first transformation handle the generation of the Table model from the UML product model. A few examples of helpers and rules of this transformation are given in the Appendix, with extensive comments in natural language.

A part of the generated spreadsheet for the scenario *BrowseCatalog* is shown in Fig. 5. 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 "low security level", the concrete value entered by the user is obtained by evaluating the expression (5.1+0.052*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 security 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.

3.3 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 [19] 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.

To consider the spreadsheets as a mark model for the transformation, we apply the same principle of separation of concerns and break the transformation into a series of simpler transformations as in [7]. Three extra model transformations have to be done before performing the actual binding: a) from the spreadsheets (XML file) to an XML model; b) from XML model to the required syntax in Ecore-based format; c) which is further extracted as an XML file that can be accepted by ATL. The main transformation to perform the actual binding takes place now, after the mark model is ready to be injected into the model transformation as an XML file with the required syntax. As an example, the helper called by different rules to get the value of an attribute is shown in the Appendix.

4. RELATED WORK

This section surveys briefly work from literature related to software performance engineering in the context of Model-Driven Architecture, where the concepts of platform-independent and platform-specific models were introduced. Special attention is given to work focused on software product lines.

The Model-Driven Architecture approach is extended in [10] with non-functional modeling and analysis concepts by adding new models and transformations for validation activities. The concepts of platform independent and platform specific are used through the new type of models to obtain an accurate validation.

The concept of performance completions was proposed in [26] to close the gap between application design models and external platform factors. Performance completions provide a means to extend the modeling constructs of a system by including the influence of the underlying platforms and execution environments in performance evaluation models.

A model transformation framework is proposed in [25] for automatically including the impact of middleware on the architecture and the performance of distributed systems. The middleware descriptions are presented as a library in the framework. Using this library, designers can model the system with different types of middleware and then obtain a platform-specific model. A LQN model build by hand is used for performance evaluation.

An approach for performance prediction of component-base software systems in proposed in [2]. The approach based on operational analysis of QN models where performance bounds are computed without deriving a QN model from the software specification. Performance bounds such as system throughput and response time are used to answer several performance-related and what-if questions such as the bottleneck resource if the platform configuration is changed.

A method for designing parametric performance completions that are independent of a specific platform is proposed in [13]. The variability in the platforms is described by using a feature model. The completions can be instantiated for different environments by explicitly coupling the transformations to performance models and implementation to add the necessary details to both.

A queueing model for the performance of Web servers is presented in [14]. The model includes the impacts of workloads, hardware/software configurations, communication protocols, and interconnect topologies. It is implemented in a simulation tool and the results are validated with results from a test lab environment.

A literature survey on approaches that address non-functional requirements (NFRs) is presented in [8]. The classification is based on software variability, requirements analysis, elicitation, reusability, and traceability as well as aspect-oriented development. Variability related to SPL is also discussed.

In the context of SPL, to the best of our knowledge, no work has been done to evaluate and predict the performance of a given product 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 [4], 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 [5] 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 [3], 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 [16]. 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 [20]. 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 [6] 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 [21] to specify performance requirements by introducing several stereotypes specific to model performance requirements such as «optional» and «alternative performance feature».

Reference [17] handles one of the problems of human interaction in the context of SPL; the decision-making process that requires humans to answer questions to configure a specific product. They propose an approach for automatically optimizing the order of questions with every answer. The optimization is done in an incremental way and in real-time.

To the best of our knowledge, ours is the first approach to generate automatically a performance model of a product from the software model of the family by a chain of model transformations. We handle the variability and commonality between the products of a family and the variability of the underlying platforms. We propose to address the performance impact of the underlying platforms as aggregated platform overheads expressed in MARTE annotations attached to the affected model elements. This will keep the model simple and still allow us to generate a performance model containing the performance effects of the platforms.

5. CONCLUSIONS

This paper is an integral part of a larger research effort to integrate performance analysis in the early phases of the development process of software product lines. Our goal is to generate automatically a performance model for a given product, which can be used to analyze its performance. Through performance analysis we can gain insight into the run-time performance characteristics and thus provide guidance for design choices early in the system development.

SPL development takes advantage of the reusability of the core assets shared among the SPL members. When integrating performance analysis in the early phases of SPL development, we take advantage of the reusability concept applied to performance annotations. Instead of annotating from scratch the UML model of every automatically derived product, we propose to annotate the SPL model once with generic performance annotations. After deriving the model of a product from the family model by an automatic transformation, the generic performance annotations need to be bound to concrete product-specific values provided by the developer.

To the best of our knowledge, our research is the first to tackle this problem. Dealing manually with a large number of performance parameters and with their mapping to each model elements, by asking the developer 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. Automating the entire process of extracting this information from a product model, generating spreadsheets, and performing the actual binding make the process of providing concrete values for performance variables more user-friendly and less error-prone. It is also more efficient and easier to repeat this process every time a new generic product model is derived from the SPL model or changes to the execution environment happen. The performance characteristics of different platforms can be measured and reused for many products executed on a variety of runtime environments. Future work will use Aspect-Oriented Modeling for including the impacts of underlying platforms by presenting each PC-feature in the PC-feature model as a generic aspect model that can be reused with different applications.

6. ACKNOWLEDGMENTS

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Centre of Excellence for Research in Adaptive Systems (CERAS).

7. REFERENCES

[1] Atlas Transformation Language (ATL), www.eclipse.org/m2m/atl

[2] Balsamo, S., Marzolla, M., and Mirandola, R., “Efficient Performance Models in Component-Based Software Engineering”, Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced, 2006.

[3] Bartholdt, J., Medak, M. and Oberhauser, R, "Integrating Quality Modeling with Feature Modeling in Software Product Lines", Proc. of the 4th Int Conference on Software Engineering Advances (ICSEA2009), pp.365-370, 2009.

[4] Belategi, L., Sagardui, G., Etxeberria, L., "MARTE mechanisms to model variability when analyzing embedded software product Lines", Proc. Of the 14th Int Conference on Software Product Line (SPLC'10), pp.466-470, 2010.

[5] Belategi, L., Sagardui, G. and Etxeberria, L., “Model based analysis process for embedded software product lines”, Proc of the 2011 Int Conference on Software and Systems Process (ICSSP '11), 2011.

[6] Benavides, D., Trinidad, P., and Ruiz-Cort´es, A., “Automated Reasoning on Feature Models”, Proc. of 17th Int. Conference on Advanced Information Systems Engineering (CAiSE), 2005.

[7] Brunelière, H., ATL Transformation Example: Microsoft Office Excel Extractor, http://www.eclipse.org/m2m/atl/atlTransformations/MSOfficeExcelExtractor/ExampleMicrosoftOfficeExcelExtractor[v00.01].pdf

[8] Chung, L., Leite, J.C. sampaio do prado, “On Non-Functional Requirements in Software Engineering”, Conceptual Modeling: Foundations and Applications, pp. 363-379, 2009.

[9] Clements, P. C. and Northrop, L. M. (2001). “Software Product Lines: Practice and Patterns”, p.608, Addison-Wesley, 2001.

[10] Cortellessa, V., Di Marco, A. & Inverardi, P., “Non-Functional Modeling and Validation in Model-Driven Architecture”, Proc of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA07), pp. 25, Mumbai, 2007.

[11] Czarnecki, K., Helsen, S., and Eisenecker, U., “Formalizing cardinality-based feature models and their specialization”, Software Process Improvement and Practice, pp. 7–29, 2005.

[12] Gomaa, H., "Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures", Addison-Wesley Object Technology Series, July 2005.

[13] Happe, J., Becker, S., Rathfelder C., Friedrich, H., and Reussner, R., "Parametric performance completions for model-driven performance prediction", Performance Evaluation, Vol.67, No.8, pp.694-716, 2010.

[14] Mei, R. D. van der, Hariharan, R., Reeser, P.K., “Web Server Performance Modeling. Telecommunication Systems (TELSYS) 16(3-4), pp. 361-378, 2001.

[15] Menasce, D., Almeida, V., and Dowdy, L., "Performance by Design: Computer Capacity Planning by Example", Prentice Hall PTR, Upper Saddle River, NJ 07458, 2004.

[16] Nguyen, Q.,"Non-Functional Requirements Analysis Modeling for Software Product Lines", Proc. of Modeling in Software Engineering (MISE'09), ICSE workshop, pp. 56-61, 2009.

[17] Nohrer, A. and Egyed, A., "Optimizing User Guidance during Decision-Making", Proc. of the 15th Int Conference on Software Product Line (SPLC'11), Munich, Germany, 2011.

[18] 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.

[19] Object Management Group, "MDA Guide Version 1.0.1", omg/03-06-01, 2003.

[20] Raatikainen, M., Niemelä, E., Myllärniemi, V., and Männistö, T., "Svamp - An Integrated Approach for Modeling Functional and Quality Variability", 2nd Int Workshop on Variability Modeling of Software-intensive Systems (VaMoS), 2008.

[21] Street, J. and Gomaa, H.,"An Approach to Performance Modeling of Software Product Lines", Workshop on Modeling and Analysis of Real-Time and Embedded Systems, Genova, Italy, October 2006.

[22] Tawhid, R. and Petriu, D.C., "Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line Model", Proc. of the 2008 ACM Int. Workshop on Software Performance (WOSP08), pp. 91-102, 2008.

[23] Tawhid, R. and Petriu, D.C., "Automatic Derivation of a Product Performance Model from a Software Product Line Model", Proc. of the 15th Int Conference on Software Product Line (SPLC'11), Munich, Germany, 2011.

[24] Tawhid, R. and Petriu, D.C., "Integrating Performance Analysis in Software Product Line Development Process", book chapter in Software Product Lines - The Automated Analysis, InTech - Open Access Publisher, 2011 (in press).

[25] Verdickt, T., Dhoedt, B., Gielen, F., and Demeester, P., "Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models", IEEE Trans. on Software Engineering, Vol. 31, No. 8, 2005.

[26] Woodside, M., Petriu, D. C., and Siddiqui. K. H., "Performance-related Completions for Software Specifications". Proc. of the 22rd Int Conference on Software Engineering, ICSE 2002, pp. 22-32, Orlando, Florida, USA, 2002.

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

Appendix

Examples of ATL rules and helpers to transform a UML product model into a Table model:

```
-- Rule Interaction2Table transforms each SD
-- in UML model to a table in Table model
rule Interaction2Table {
        from    interaction : UML!Interaction
                (interaction.hasStereotype('GaAnalysisContext'))

-- Define the headers' names
        using { titles_name : Sequence(String) =
                Sequence{'Element_Type','Stereotype_Name',
                'Attribute_Name','Element_Name',
                'PC-feature_Name','Guideline for Value',
                'Generic_Parameter', 'Concrete_Value'}; }
        to      table : Table!Table(
                name <- interaction.name,
                rows <- Sequence{title_row, blank_row,

-- create a row for each attribute
                Sequence{UML!Message.allInstances()->
                    collect(e |thisModule.resolveTemp
                    (e,'hostDemand_row'))},
                Sequence{UML!Message.allInstances()->
                collect(e | thisModule.resolveTemp
                    (e,'msgSize_row')) },
                Sequence{UML!Message.allInstances()->
                    collect(e | thisModule.resolveTemp
                    (e,'commTxOvh_row')) },
                Sequence{UML!Message.allInstances()->
                    collect(e | thisModule.resolveTemp
                    (e,'commRcvOvh_row')) }} ),
-- create the title row
            title_row : Table!Row(
            cells <- Sequence{ title_cols }),
            title_cols : distinct Table!Cell
            foreach(name in titles_name)
            (content <- name),

-- create a blank row
            blank_row : Table!Row(
            cells <- Sequence{ blank_cols }),
            blank_cols : Table!Cell(
            content <- '' ") }

-- Rule Message2Rows collects all the generic tagged values
-- of the stereotypes «PaStep» or «PaCommStep» extending
-- model elements of type message and transforms them to a
-- row in a table
rule Message2Rows {
        from    message : UML!Message
        using { hostDemand_name : Sequence(String)=
                Sequence{'Message','PaStep','hostDemand',
                message.name,'Application-Annotation',
                message.getTagValues('PaStep','hostDemand')};
                msgSize_name : Sequence(String) =
                Sequence{'Message','PaCommStep','msgSize',
                message.name,

-- call helper "pcFeatureName" to get PC-feature
-- affects attribute msgSize
            message.pcFeatureName('PaCommStep','msgSize'),

-- call helper "getTagValues" to get the generic attribute value
            message.getTagValues('PaCommStep','msgSize')};
            commTxOvh_name : Sequence(String) =
            Sequence{'Message','PaCommStep','commTxOvh',
            message.name,
            message.pcFeatureName('PaCommStep','commTxOvh'),
              message.getTagValues('PaCommStep','commTxOvh')};
              commRcvOvh_name : Sequence(String) =
             Sequence{'Message','PaCommStep','commRcvOvh',
             message.name},
          message.pcFeatureName('PaCommStep','commRcvOvh'),
         message.getTagValues('PaCommStep','commRcvOvh')};}
            to   hostDemand_row : Table!Row(
                        cells <- Sequence{hostDemand_cols}),
```

```
        hostDemand_cols : distinct Table!Cell
                foreach(name in hostDemand_name)
                content <- name),
        msgSize_row : Table!Row(
                cells <- Sequence{ msgSize_cols }),
        msgSize_cols : distinct Table!Cell
                foreach(name in msgSize_name) (
                content <- name),
        commTxOvh_row : Table!Row(
                cells <- Sequence{ commTxOvh_cols}),
        commTxOvh_cols : distinct Table!Cell
                foreach(name in commTxOvh_name) (
                content <- name),
        commRcvOvh_row : Table!Row(
                cells <- Sequence{ commRcvOvh_cols}),
        commRcvOvh_cols : distinct Table!Cell
                foreach(name in commRcvOvh_name) (
                content <- name)}

-- This helper returns the tagged value of the
-- stereotype's attribute; both stereotype and
-- attribute name are given as parameters
        helper context UML!Element  def :
                getTagValues(stereotype:String,tag:String) :
                UML!Element =
        if      self.getAppliedStereotypes()->
                select(e | e.name =stereotype)->notEmpty()
        then self.getValue(self.getAppliedStereotypes()
                ->select(e|e.name=stereotype )->first(),tag)
                ->first()
        else "      endif;

-- This helper returns "true" if the respective model element is
-- stereotyped with the stereotype name given as a parameter
        helper context UML!Element  def:
                hasStereotype(stereotype:String) :
                Boolean =  self.getAppliedStereotypes()
                -> exists(c|c.name.startsWith(stereotype));
```

```
-- This helper returns the PC-feature name affecting the
-- respective attribute;both stereotype and attribute name are
-- given as parameters
        helper context UML!Element def :
                pcFeatureName(stereotype:String, name:String):
                String =
        if      self.getAppliedStereotypes() ->
                select(e | e.name = stereotype)->notEmpty()
        then UML!Class.allInstances() ->
                select(class|class.getTagValues
                ('pc-feature','AttList')=name) ->
                collect(c|c.name)->first()
        else "          endif;
```

An Example of a helper from the transformation performing the actual binding:

```
-- This helper returns the value of the attribute 'value' and gets
-- as a parameter the value of the attribute 'name' by checking
-- all elements in mark model 'parameters'
        helper def : getParameter (variable : String) : String =
                XML!Element.allInstancesFrom
                ('parameters')->select(m|m.name = 'Row')->
                select(a|a.getStringAttrValue('name')= variable)
                ->first().getStringAttrValue('value');

-- This helper is called by the previous one to return the value
-- of a string attribute. It returns an empty string if the attribute
-- doesn't exist.
        helper context XML!Element def:
                getStringAttrValue(attrName : String) : String =
        let   attX :Sequence(XML!Attribute)= self.children
                ->select(a|a.oclIsTypeOf(XML!Attribute) and
                a.name = attrName)->asSequence()  in
        if      attX -> notEmpty()
        then    attX ->first().value
        else "              endif;
```