4

Integrating Performance Analysis in Software Product Line Development Process

Rasha Tawhid and Dorina Petriu
Carleton University
Canada

1. Introduction

A Software Product Line (SPL) is a set of similar software systems that share a common set of features satisfying a particular domain, and are built from a shared set of software assets using a common means of production. Experience shows that by adopting a SPL development approach, organizations achieved increased quality and significant reductions in cost and time to market [Clements & Northrop, 2001].

Model-Driven Development (MDD) is a well-known paradigm that aims at capturing every important aspect of software development through models. An emerging trend apparent in the recent literature is that the SPL development moves toward adopting a MDD paradigm, which means that models are increasingly used to represent software artifacts of the family or of individual products [Groher & Voelter, 2009]. MDD plays an important role in the verification of non-functional properties (such as performance, reliability, security) of UML software models extended with information specific to the property to be evaluated [Woodside et al., 2005]. UML software models can be annotated with performance properties by using the *UML Performance Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)* [OMG, 2011] recently standardized by OMG.

This chapter presents a comprehensive methodology for integrating performance analysis in the early phases of SPL model-driven development process, whose goal is to evaluate the performance characteristic of different products by generating and analyzing quantitative performance models [Tawhid & Petriu, 2008a, 2008b]. We start by adding generic performance annotations expressed in MARTE to the UML model representing the set of core reusable SPL assets. A model transformation realized in the Atlas Transformation Language (ATL) derives the UML model of a specific product with concrete MARTE performance annotations from the SPL model. The product derivation process binds the variability expressed in the SPL to a specific product, and also the generic SPL performance annotations to concrete values provided by the designer for this product. The proposed model transformation approach can be applied to any existing SPL model-driven development process using UML for modeling software.

It is known that one of the main concepts of software product line development is to take advantage of the reusability of the set of core assets shared among the members of a family of products, instead of building each product from scratch. In this work, we apply the same

reusability concept to the performance annotations, by integrating software performance engineering techniques in the early phases of SPL development. Instead of annotating from scratch each UML model of each product, we propose to annotate the SPL model once with generic annotations, and to provide binding information when deriving the annotated model of a desired product from the generic SPL model.

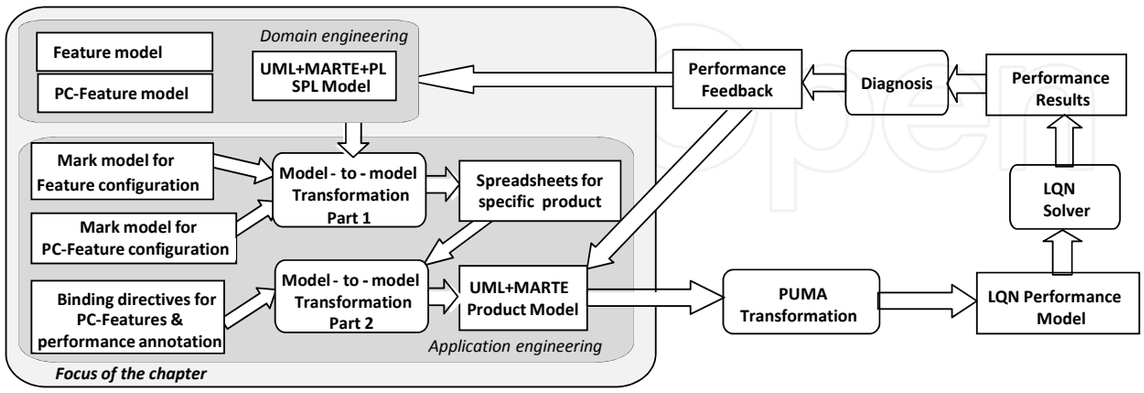

Fig. 1. Approach for deriving a product performance model

The objective of the research presented in this chapter is to automatically generate a performance model for a given product from a performance-annotated SPL model. The main research challenge originates from the mismatch between the meanings of the two models. While a SPL model is a set of core "generic" asset models that are building blocks for many different products with all kind of options and alternatives, a performance model is an instance-based representation of a runtime system, focusing on how the system is using available resources and how competition for resources impacts the system performance (response time, throughput, utilization, etc.) The derivation of a performance model requires two model transformations, as shown in Fig.1: a) from the annotated SPL model to a product model with performance annotations, and b) from the outcome of the first step to a performance model. The work presented here focuses on the first transformation as illustrated by the shaded area in Fig.1, whereas the second transformation for deriving automatically a Layered Queueing Network (LQN) performance model for a specific product applies the PUMA transformation approach previously developed in our research group [Woodside et al., 2005].

The automatic derivation of a specific product model based on a given feature configuration is enabled through the mapping between features from the feature model and their realizations in the design model. In this chapter, an efficient mapping technique is used, which aims to minimize the amount of explicit feature annotations in the UML design model of SPL. Implicit feature mapping is inferred during product derivation from the relationships between annotated and non-annotated model elements as defined in the UML metamodel [Tawhid & Petriu, 2011a].

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 (obtained 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 factors [Woodside et al., 2002]. 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 introducing a so-called Performance Completion feature (PC-feature) model, which characterizes 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 [Tawhid & Petriu, 2011b]. Performance model helps software developers explore various design alternatives. It also addresses the problem of domain evolution arising when an existing product runs on a new platform. In this chapter, we explain how this evolution can be propagated to the performance model through the PC-feature model.

The chapter is organized as follows: section 2 discusses related work; section 3 presents the domain engineering process where the SPL model and two different kinds of feature models are created; the model transformation approach for generating a given product model is illustrated with a case study in section 3; section 4 analyzes the performance effects of different security levels for communication channels running on two different architectures; and section 5 presents the conclusions.

2. Related work

This section presents related research on product derivation approaches and different feature mapping techniques. Work related to performance analysis of software system, addressing quality attributes in SPL is also discussed.

Voelter et al. propose an approach that integrates aspect-oriented (AOSD) and model-driven software development (MDSD) techniques to support variability management and product derivation [Groher & Voelter, 2009]. Two different ways of dealing with variability are identified: a) negative variability which selectively removes parts of a model based on the presence or absence of features in the configuration model; b) positive variability which starts with a minimal core of common SPL artifacts and selectively adds additional product-specific parts through model weaving. Our approach applies a similar concept of positive variability through automatic model transformation. In [Stoiber & Glinz, 2009], aspect-orientation is combined with table-based modeling by using the ADORA modeling language. An approach for deriving the architecture of a product by selectively copying elements from the SPL architecture (which covers all possible product aspects) based on a product-specific feature configuration is proposed in [Botterweck et al., 2009]. This approach is concerned only with the derivation of the high-level product architecture, while our approach derives both the structural and behavioural views of the product design model. An Eclipse-based tool called FeatureMapper that defines the mapping of features in the problem space to model elements realizing these features in the solution space is proposed in [Heidenreich et al., 2007, 2008]. The set of selected features for a product combined with the mapping model are interpreted by the FeatureMapper transformation component to derive a product model.

An approach for expressing variability in a family model based on a *feature-based model template* by mapping features to model elements realizing them is introduced in [Czarnecki et al., 2005a, 2005b]. Each model element is annotated with a presence condition (PC), indicating whether the element should be present in a template instance or not. The model template is automatically instantiated by evaluating the PCs for a given feature configuration. The concept of negative variability is applied, by removing model elements whose PC evaluates to false. A drawback of this approach is that the model template is cluttered with variability specifications for each model element. Some issues related to the behavioural derivation of a given product model are discussed in [Istoan et al., 2011]. It is shown that the composition order is significant when using Aspect-Oriented modeling, since different orders for composing sequence diagrams leads to different derived products.

A model-driven approach for SPL evolution is proposed in [Gamez & Fuentes, 2011], which automatically propagates the evolution changes of a cardinality-based FM into existing configurations.

The Product Line UML-Based Software Engineering (PLUS) method introduced in [Gomaa, 2005] provides several concepts and stereotypes to express variability in multiple views of SPL. The mapping between features and the model elements realizing them is introduced through a separate tabular representation of feature/use case and feature/class relationships. Our approach introduces a different mapping technique by annotating each class and use case with the feature(s) requiring it. The automatic derivation of a concrete product from the SPL model according to a set of chosen features is not addressed in [Gomaa, 2005]. The PLUS method is extended in [Street & Gomaa, 2006] to specify performance requirements by introducing several stereotypes specific to model performance requirements such as «optional» and «alternative performance feature». Although feature modeling is essential in SPL, the concept of "feature" is not a first-class model element in UML. In order to overcome this problem, different stereotypes for representing features and feature dependency have been defined in literature (however, none is standard yet). Our variability profile is based on Gomaa's work, especially on PLUS [Gomaa, 2005]. However, our approach has the following main differences from PLUS: a) we proposed an automatic derivation of a product model from a SPL model; b) we deal with MARTE performance annotations, both in the source and target models; c) we use sequence diagrams for behaviour representation taking advantage of their enhanced modeling power,; d) we introduce variability within a sequence diagram through *Combined Fragments*; e) we introduce the so-called *Performance Completion feature model*.

Several works have been done on performance analysis addressing quality attributes in SPL. A method for designing parametric performance completions that are independent of a specific platform is proposed in [Happe et al., 2010]. 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.

Model-driven development and SPL paradigms are integrated together to model embedded software systems in [Belategi et al., 2010a]. An analysis method taking into account scenarios, platform, and variability for embedded SPL has been proposed. Although the authors consider the SPL architecture as a critical asset for representing quality attributes and their compliance to quality goals, they have not addressed how quality attributes are

modeled in the architecture. In [Belategi et al., 2010b], 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 [Belategi et al., 2011] 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 [Bartholdt et al., 2009] 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. An approach called Svamp is proposed to model functional and quality variability at the architectural level of the SPL [Raatikainen et al., 2008]. 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. The Model-Driven Architecture approach is extended in [Cortellessa et al., 2007] 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.

To the best of our knowledge, in the context of SPL, no work has been done previously to evaluate and predict the performance of a given product by generating a formal performance model. Most of the existing work aims to model non-functional requirements (NFRs) in the same way as functional requirements. The related works mentioned above are concerned with the interactions between selected features and the NFRs and propose different techniques to represent these interactions and dependencies.

3. 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 products that are family members from reusable artifacts created in the first phase, instead of starting from scratch.

The domain engineering process is a development cycle *for* reuse and includes, but is not limited to, creating the requirement specifications, domain models , architecture, reusable software components [Clements & Northrop, 2001].

The SPL assets created by the domain engineering process which are of interest for our research are represented by a multi-view UML design model of the family, called the *SPL model*, consisting of a superimposition of all variant products. The creation of the SPL model employs two separate UML profiles: a *product line* profile for specifying the commonality

and variability between products, and the MARTE profile for performance annotations. Another important outcome of the domain engineering process is the feature model used to represent commonalities and variabilities between family members in a concise taxonomic form. Additionally, the PC-feature model is created to represent the variability space of the performance completions.

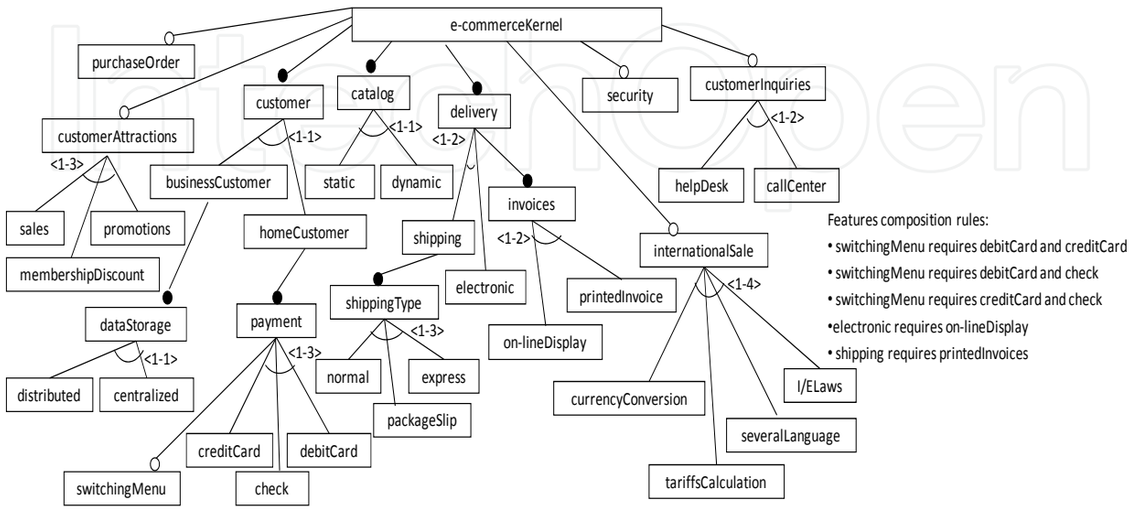

Fig. 2. Feature model of the e-commerce SPL

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

3.1 Feature model

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. The feature model of an e-commerce SPL is represented in Fig. 2 in the extended FODA notation, Cardinality-Based Feature Model (CBFM) [Czarnecki et al., 2005]. However, this diagram is represented in the source model given as input to 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 *BusinessCustomer* and *HomeCustomer* are mutually exclusive features and hence they are grouped into an *exactly-one-of* feature group called *Customer*. While, the three optional features *CreditCard*, *DebitCard*, and *Check* are grouped into an *at-least-one-of* feature group called *Payment*. Thus, an individual system can provide at least one of these features or any number of them. In the case of an individual system providing all of these features, the user can choose one of them during the run-time execution. In addition to functional features, we add to the diagram another type of features

characterizing design decisions that have an impact on the non-functional requirements or properties. For example, the architectural decision related to the location of the data storage (centralized or distributed) affects performance, reliability and security, and is represented in the diagram by two mutually exclusive quality features. This type of feature related to a SPL design decision is part of the design model, not just a platform-related PC-feature required only for performance analysis.

The regular feature model represents the set of all possible combinations of features for the products of the family, describing 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 the feature configuration based on the product's requirements. To enable the automatic derivation of a given product 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.

3.2 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.

The functional requirements of the SPL are modeled as use cases shown in Fig. 3. The kernel use cases required by all the family members are shown in white, the optional use cases that may be used by any member are drawn in light grey, and the alternative use cases used only by some members are shown in dark grey. In order to avoid polluting our model with extra annotations and to ensure the well-formedness of the derived product model, we propose to annotate explicitly the minimum number of model elements within each diagram of our SPL model. For instance, in the use case diagram, only the optional and alternative use cases are annotated with the name of the features requiring them (given as stereotype attributes); since a kernel use case represents commonality, it is sufficient to just stereotype it as «kernel». Other model elements, such as actors, associations, generalizations, properties, are mapped implicitly to feature through their relationship with the use cases, so there is no need to clutter the model with their annotations. The evaluation of implicit mapping during product derivation is explained in section 4. The structural view of the SPL is presented as a class diagram; Fig. 4 depicts a small fragment. The classes that are common to all members of the SPL are stereotyped as «kernel». The variability that distinguishes the members of a family from each other is explicitly modeled by classes stereotyped as «optional» or «variant»; such classes 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.

In cases where a class behaves differently in different product (such as *CustomerInterface* in B2B and B2C systems) a generalization/specialization hierarchy is used to model the different behaviours of this class. The two subclasses *B2BInterface* and *B2CInterface* are used by B2B systems and B2C systems, respectively. The same happens with the superclass *SupplierInterface*, which is specialized into two variants *POSupplier* and *Supplier*.

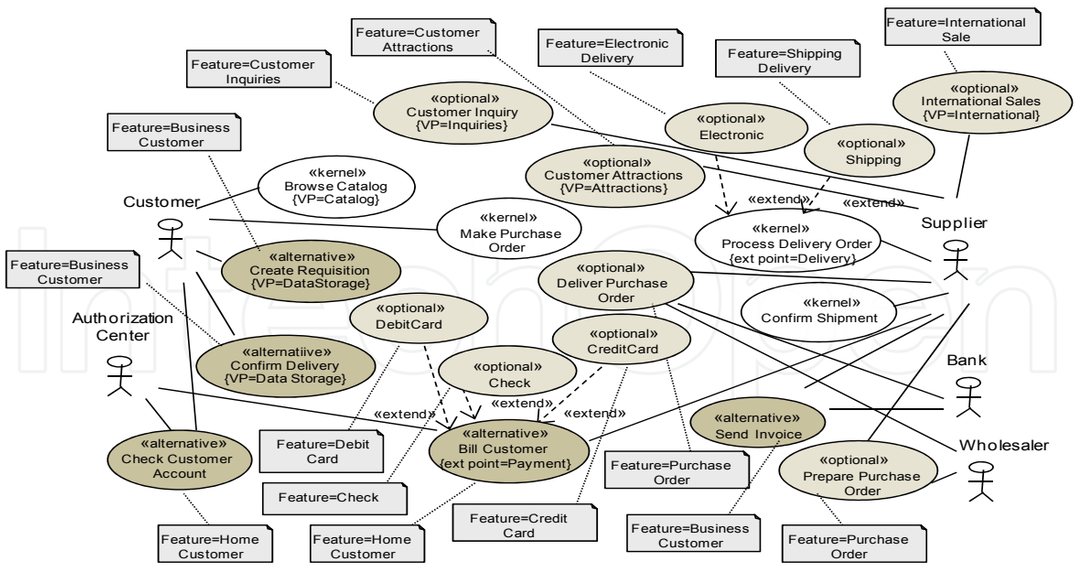

Fig. 3. Use case model of the e-commerce SPL

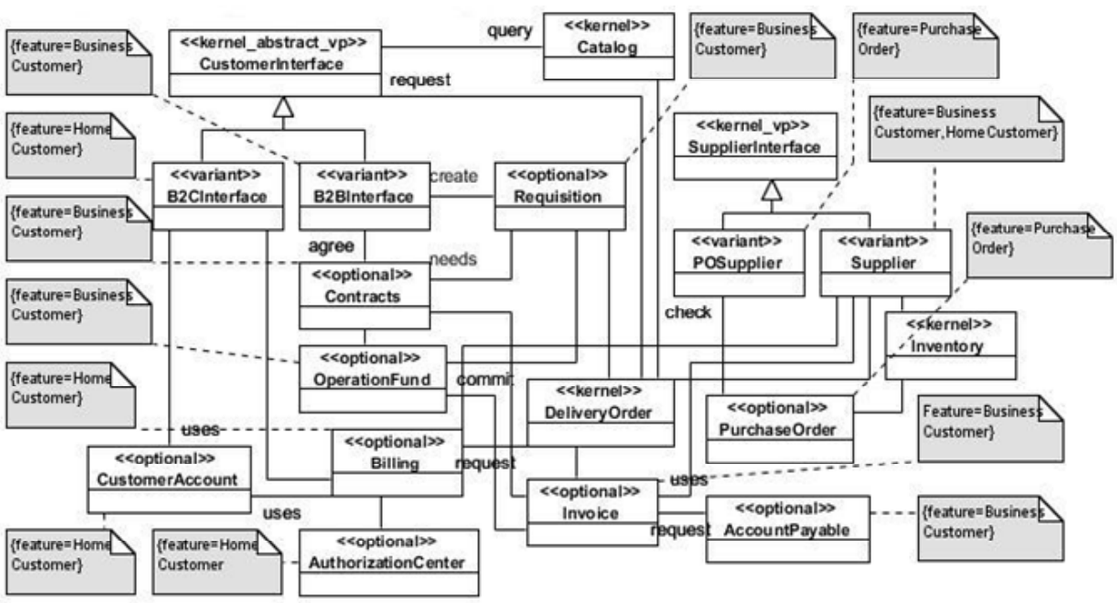

Fig. 4. A fragment of the class diagram of the e-commerce SPL

The behavioural SPL view is modeled as sequence diagrams for each scenario of each use case. Fig. 5 illustrates the alternative scenario *Create Requisition*. Variability in the sequence diagram may be expressed by using *alt* or *opt* fragments stereotyped as «variation point». For example, the *alt* fragment stereotyped with «variation point» {vp=*Data Storage*} gives two alternative choices based on the value of the Data Storage feature (Distributed or Centralized). The stereotypes in Fig. 5 are MARTE performance annotations [OMG, 2011]. «GaAnalysisContext» is a stereotype indicating that the entire interaction diagram is to be considered for performance analysis. Each lifeline is stereotyped as «PaRunTInstance», providing an explicit connection at the annotation level between a role in a behavior

definition (a lifeline) and a runtime instance of a process or thread (active object). For example, the tag {instance= CBrowser} indicates which runtime instance of a process executes the lifeline role, while the tag {host=$CustNode} indicates the physical node from the deployment diagram on which the instance is running, given by the variable $CustNode. (For convenience, we use names starting with '$' for all MARTE variables). 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. 5, the message *requisitionRequest* is stereotyped as an execution step:

«PaStep» {hostDemand = ($ReqSD,ms), respT = (($ReqT,ms, percent95), calc)}

where hostDemand indicates the execution time required by the step, given by the variable $ReqSD in time units of milliseconds. The same message *requisitionRequest* is also stereotyped as a communication step:

«PaCommStep» { msgSize = ($MReq,KB)}

where the message size is the variable $MReq in KiloBytes. 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 workload of a scenario is defined as a stream of events driving the system; a workload may be open or closed. In our example the workload is closed with a number of users $N1 and user think time for a user $Z1:

«GaWorkloadEvent» {pattern=(closed (population=$N1),(extDelay=$Z1))}

3.3 Performance completions

In SPL, different members may vary from each other in terms of their functional requirements, quality attributes, platform choices, network connections, physical configurations, and middleware. Many details contained in the system that are not part of its design model, but still affecting the performance at run-time, need to be added to the performance model. Performance completions, as proposed by Woodside [Woodside et al., 2002], are a manner to add platform details, closing 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 environments in performance models.

Since performance analysis depends on the software to hardware allocation, another structure diagram that is not usually represented in SPL models has to be provided in our approach. The deployment diagram for the SPL is built assuming maximum distribution, which means providing the largest number of processors that might ever be used for any product of the SPL. However, it doesn't mean providing a processor for every single artifact manifesting an instance of an active or passive class. If it is known that some instances have to run always on the same processor, they will be co-allocated on the same node. The

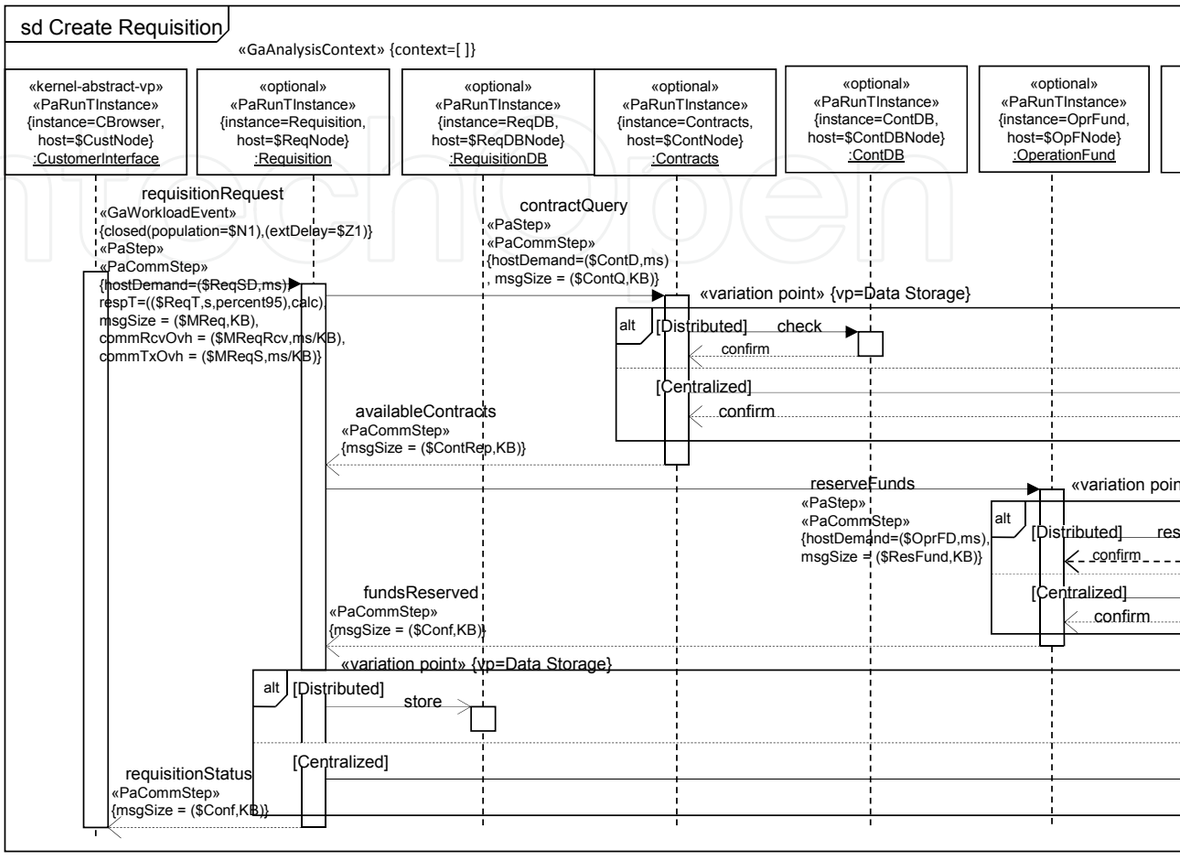

Fig. 5. SPL Scenario Create Requisition

deployment diagram contains all the possible artifacts contained in all the products, even artifacts corresponding to optional or variant classes. During the domain engineering process for our case study, two different deployment diagrams for the SPL system are provided, distributed and centralized, corresponding to the two alternative architectures.

This section covers the variability space of the performance completions and represents it through the Performance Completion feature model (PC-feature model) shown in Fig. 6. Each feature from the PC-feature model 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) it affects, which are MARTE stereotype attributes associated to 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.

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 two security protocols: Secure Socket Layer (SSL) and Transport Layer Security (TLS) that can be augmented to the applications. Furthermore, we introduce 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 [Menasce et al., 2004]. Each security level requires different extra times for sending and receiving secure messages. These overheads are mapped to the communication overheads in the deployment diagram through the attributes *commRcvOvh* and *commTxOvh*, which represent the host demand overheads for receiving and sending messages, respectively.

Each type of communication channel has different capacity for the amount of information that can be transmitted over this channel. As the channel's capacity increases, the time for data transmitted over this 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 respectively mapped to the attributes *capacity* and *blockT* stereotyping each communication node in the deployment diagram.

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

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

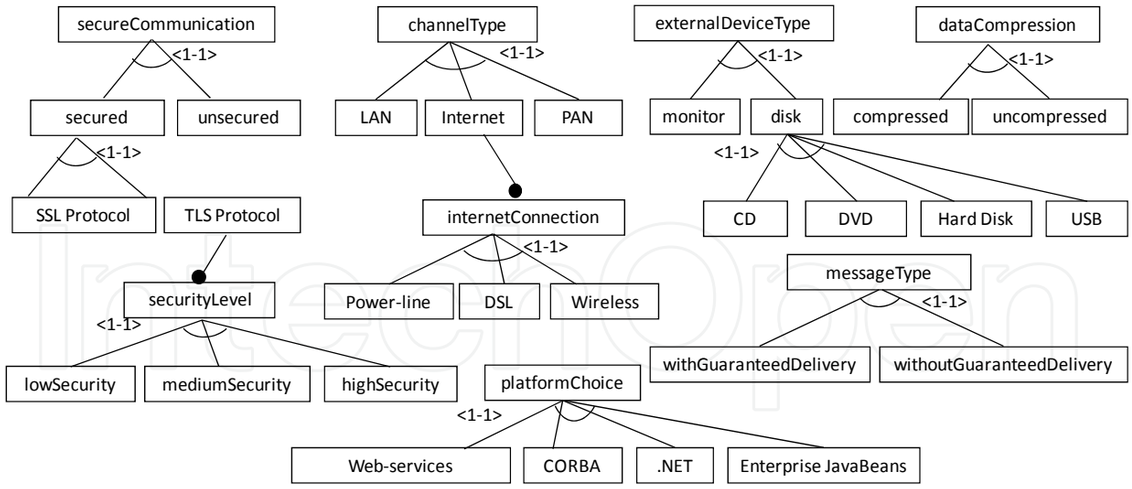

Fig. 6. Part of the Performance-Completion feature model of the e-commerce SPL

Data compression requires extra operations which 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 reduces the amount of data to be transferred and decreases the delivery time (e.g., a compression algorithm may reduce the size of data to 60% [Happe et al., 2010]). Thus, the amount of compressed data transmitted over a physical channel is mapped to the performance attribute message size through the attribute *msgSize* of a stereotype «PaCommStep» annotating a communication step in the sequence diagram. Similarly, the delivery time of a message may vary if the communication is with or without guaranteed delivery [Happe et al., 2010], which affects the attribute *commTxOvh*.

Mapping a platform independent to a platform specific model has an impact on the system performance. The PC-feature group *platformChoice* includes different alternative types of middleware such as CORBA, Web-services, etc., which will affect also the communication overheads.

MARTE provides specifically the concept of "external service calls" to represent resources that are not explicitly modeled within the UML design model, but may have an impact on performance. Examples of such external calls are disk operations hidden in database calls. The feature *externalDeviceType* represents different choices of storage devices, such as disk and monitor and different disk types. Each device has different speed to "read" and "write" a block of data. These features are mapped to the service time of external resources through the attribute *externalOpCount* stereotyping an execution step.

It is important to note that some of the performance-affecting attributes are contained directly in the MARTE annotations in the design model. For instance, the message size corresponding to a message from a sequence diagram may be indicated by the attribute *msgSize* of the stereotype «PaCommStep» extending the message. Similarly, CPU execution times of different scenario steps are indicated by the attribute *hostDemand* of the stereotype «PaStep». The product model obtained by the transformation presented next will include

both the performance attribute contained directly in the design model and the platform factors corresponding to PC-features.

4. Model transformation approach

The automatic derivation of a concrete product model based on a given feature configuration is enabled through the mapping between features from the feature model and their realizations in the design model of the SPL. In this section, we present an efficient mapping technique that aims to minimize the amount of explicit feature annotations in the UML design model of SPL. The product model corresponding to the desired feature configuration is instantiated automatically through a model-to-model transformation, where the transformation process evaluates the SPL model elements' annotations for the selected feature configuration. The model transformation process and its implementation in ATL are presented as well.

4.1 Mapping technique

Modeling variability in SPL models can be achieved in different ways: 1) annotating different diagrams of the reusable SPL model with variability specifications mapping features from the feature model to model elements realizing them; and 2) using a separate model for variability that can be linked to different model elements of the reusable SPL model. In our work, we apply the first approach by using a product line (PL) profile similar to [Gomaa, 2005]. We are aiming to annotate the UML model of SPL with a minimum amount of variability specifications.

The annotation approach has a number of advantages over the separate variability modeling: a) model elements subject to variability are clearly noticeable; b) the consequence of selecting a feature is directly shown on the design model; c) the mapping is easier to retrace and understand; and d) the expressive capability is enhanced. However, a significant drawback of the annotation approach that makes it error-prone is the fact that the SPL models become cluttered with variability specifications, which becomes worse as models grow in size and complexity.

The annotation approach proposed in this research mitigates this drawback by reducing the type and number of explicitly annotated model elements as much as possible. The decision what types of elements to annotate explicitly depends on the application domain and should be taken early in the domain engineering process. The mapping of features to non-annotated model elements is implicit, and can be inferred from their relationships with annotated model elements. Such relationships are defined in the UML metamodel and are explored in the transformation rules during product derivation by navigating the model according to the UML metamodel and well-formedness rules. For instance, in a class diagram of the SPL reusable model, we annotate explicitly the variability of classes with the names of the features requiring each class, but leave the associations without variability annotations. The unspecified mapping of features to each association can be inferred from the annotations of the two classes connected to the association ends. Thus, the mapping of features to classes is explicit and that of features to associations is implicit. Whenever a model element is not explicitly annotated with corresponding feature(s) through a stereotype or its attributes in the SPL model, the automatic transformation process needs to decide whether to copy this

element to the target model or not. This decision is based on several factors: a) the type of this non-annotated element; b) the specifications and well-formedness constraints of the modeling language; c) the presence or absence of other annotated elements related to it; d) the containment hierarchies defined in the metamodel; e) the cardinality of this element.

For example, according to the UML metamodel, a binary association has to be attached to a classifier at each end. Therefore, the decision whether a binary association has to be copied or not to the target is based on the selection of both of its classifiers. The binary association is created in the target model if and only if both of its *memberEnd* properties have their classifiers already selected and created. At the same time, if only one of its classifier is selected and created in the target model, the property attached to this unselected association and owned by the selected classifier should not be created in the target model. The interpretations of the implicit mapping will be explained in more detail in the description of the transformation rules.

The proposed mapping technique ensures that the derived product model is a well-formed model by enforcing the well-formedness constraints during the transformation process. Each time a new model element is selected and added to the target model, the verification of its well-formedness rules is guaranteed by construction, according to the transformation rules that are based on the UML metamodel.

4.2 Model transformation process

Our model transformation approach takes as input the SPL source model created during the domain engineering process in section 3 and generates a product target model for a given member of the SPL. The model transformation consists of two parts as shown in Fig. 1. The first part generates binding directive spreadsheets, asking the user to enter concrete values for all generic performance annotations and platform allocations for the given product, while the second part takes as input the spreadsheets with the concrete values provided by the user and generates a specific product model with concrete performance annotations that is deployed on concrete resources and is running on a specific platform. As mentioned before, our model transformation approach applies the concept of positive variability where we start by selecting and copying the SPL model elements that represent kernel features to the target model, then selectively add other elements realizing the desired optional and alternative features; all this is realized by a model transformation approach described below.

The product derivation process is initiated by specifying a given product through its feature configuration (i.e., the legal combination of features characterizing the product). The selected features are checked for consistency against the feature dependencies and constraints in the feature model, in order to identify any inconsistencies. An example is checking to ensure that no two mutually exclusive features are chosen. The feature configuration is considered a parameter for the transformation, which should be set without editing the source model. The second step in the derivation process is to select the use cases realizing the chosen features. All kernel use cases are copied to the product use case diagram, since they represent functionality provided by every member of the SPL. If a chosen feature is realized through extend or include relationships between use cases, both the base and the included or extending use cases have to be selected, as well. A use case containing in its scenario variation point(s) required to realize the selected feature(s) has to be chosen, too. The optional and alternative use cases are

selected and copied to the target use case diagram if they are mapped to a feature from the feature configuration. The interpretations of other non-annotated elements will be explained in the description of the transformation rules. Finally, the use case diagram for the product is developed after all the PL variability stereotypes were eliminated. The third step is to derive the product class diagram by selecting first all kernel classes from the SPL class diagram. Optional and variant classes needed for the desired product are selected next (each is annotated with the feature(s) requiring it). Moreover, superclasses of the selected optional or variant classes have to be selected as well. The PL variability stereotypes are not copied to the target model. An association between two classes is copied to the target model if and only if both classes are selected.

The SPL deployment diagram has to be tailored to the concrete product in the fourth step. One of the two types of the deployment diagrams (centralized or distributed) has to be chosen based on the mutually exclusive feature group *DataStorage*. For instance, the centralized architecture is chosen and copied to the target model if the feature *centralized* is selected. The final step of the first part in our transformation approach is to generate the sequence diagrams corresponding to different scenarios of the chosen use cases. Each scenario of a chosen use case is recognized through a sequence diagram which has to be selected from the source model and copied to the target one.

The PL variability stereotypes are eliminated after binding the generic roles associated to the life-lines of each selected sequence diagram to specific roles corresponding to the chosen features. For instance, the sequence diagram *Create Requisition* has the generic alternate role *CustomerInterface* which has to be bound to the concrete role *B2BInterface* to realize the feature *BusinessCustomer*. However, the selection of the optional roles is based on the corresponding features. For instance, the generic optional role *CustomerDB* is selected if the feature *Centralized* data storage is chosen.

The mapping between the PC-features and performance attributes takes place during the first part of the model transformation and requires user input. The transformation extracts all the information needed for the mapping from the annotated product model and the PC-feature model, and generates spreadsheets for the given product. The second part of the model transformation takes as an input the spreadsheets with the values for bindings directives provided by the user, and produces the given product model with concrete values for performance annotations.

4.3 ATL implementation of the proposed approach

This subsection presents the implementation of the model transformation described in the previous subsection in the Atlas Transformation Language (ATL) [ATL], which is specialized for model transformations. The source model is the SPL model described in section 3 with two profiles applied, MARTE and PL, and the target model is that of a particular product. The transformation rules handle the implicit and explicit mapping of features to SPL design models. The 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. A few examples of ATL transformation rules are given bellow, with extensive comments in natural language.

We need to create in the target model all the model element types that compose a class diagram according to the UML metamodel: *Class*, *Property*, *Operation, Generalization, and Association* [OMG, 2007]. Since an optional or alternative class is annotated with the feature(s) requiring it, the class element is selected if and only if the feature given in its annotation is present in the feature configuration. The following rule is applied to each model element of type *Class* from the source model, checking whether to select and copy it to the target model. We need to distinguish between a property representing an attribute (related to the class by *ownedAttribute*) and a property representing an association end (related to an association by *memberEnd*). A property representing an attribute has to be selected if its container is selected. However, the one representing an association end is selected if and only if its class container and the related association are selected.

In order to select and copy to the target model only the associations between selected classes as well as their *memberEnds*, we have to navigate from the property of a selected class that represents an association end to the other end of the association and check whether the class on this end is selected or not. Assume that there are two classes: *ClassA* and *ClassB* connected with an association *AB*. *ClassA* owns a property *PA1* that has an attribute *type* referencing the other end of the association, *ClassB*. In turn, *PA1* has an attribute association referencing the association *AB*. The rule that interprets this implicit mapping navigates from the selected *ClassA* to the other end *ClassB* though the attribute *type* of the property *PA1* and checks whether *ClassB* is selected or not. If *ClassB* is selected, property *PA1* is selected as well. Last step is to navigate through the attribute *association* of property *PA1* to the association *AB*, and to copy it to the target model.

```
        -- Rule Class checks each model element of this type whether to copy it
        -- to the target model by calling the helper selectedElement()
rule Class {
        from
                s : UML! Class (s.selectedElement())
        -- Copying the class and checking for each property representing an
        -- association end whether the class on the other end of the association
        -- is selected or not
        to
                t : UML!Class(name <- s.name, ownedAttribute <-
                 s.ownedAttribute->select(e|e.type.selectedElement())

        -- Whenever the class on the other end is selected, the property
        -- representing a memberEnd is copied to the target model by calling
        -- the lazy rule Property
                 -> collect(e|thisModule.Property(e)),

        -- Copying the property owned by the class by calling the lazy rule
        -- Attribute
                 ownedAttribute<-s.ownedAttribute->select(e|e.association
                 ->oclIsUndefined())->collect(e|thisModule.Attribute(e)),

        -- Copying the operation owned by the class by calling the lazy rule
        -- Operation
```

```
                ownedOperation <- s.ownedOperation ->
                 collect(e | thisModule.Operation(e)),
        -- Copying the generalization owned by the class by calling the lazy rule
         -- Generalization
                        generalization <- s.generalization
                        -> collect(e | thisModule.Generalization(e))) }
        -- This helper returns “true” if the respective element is selected by
        -- checking whether the tagged value of its stereotype’s property
        -- existed in the feature configuration
        Helper context UML!Elementdef: selectedElement() : Boolean = if
        self.hasStereotype('kernel')
         or UML!Class.allInstances()->
         select(class | class.getTagValue('optionalfeature','selected')
         ='true'
         Or class.getTagValue('alternativefeature','selected')
         =‘true')-> collect(c | c.name)->
         includes(self.getTagValue('variant','feature')
         or self.getTagValue('optional','feature') or
         self.getTagValue('alternative','feature'))
        then true else false
        endif;
        -- This helper returns “true” if the respective model element is
         -- stereotyped with the stereotype name given as a parameter
        Helper context UML!Elementdef: hasStereotype(stereotype:String)
         :Boolean = self.getAppliedStereotypes()->
         exists(c | c.name.startsWith(stereotype));
        -- This helper returns the tagged value of a stereotype’s property both
         -- stereotype and property name are given as parameters
        Helper context UML!Elementdef:getTagvalue
        (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)
        else OclUndefined
        endif;
        -- This lazy rule is executed when called by the previous rule to copy an
         -- ownedAttribute property with its upper, lower, and default
         -- multiplicity values to the target model
lazy rule Attribute{
        from
                s : UML!Property
        to
                t : UML!Property(name <- s.name, type <- s.type,
         upperValue<-thisModule.LiteralUnlimitedNatural(s.upperValue),
         lowerValue <- thisModule.LiteralInteger(s.lowerValue),
         defaultValue <- thisModule.LiteralString(s.defaultValue)) }
```

```
        -- This lazy rule is called by the previous rule to copy a memberEnd
        -- property with its upper and lower multiplicity values to the target
        -- model as well as copy the association attached to it by calling the
        -- lazy rule Association
lazy rule Property{
        from
                s : UML!Property
        to
                t : UML!Property(name <- s.name, type <- s.type,
                association <- thisModule.Association(s.association),
        upperValue<-thisModule.LiteralUnlimitedNatural(s.upperValue),
        lowerValue <- thisModule.LiteralInteger(s.lowerValue)) }
        -- This lazy rule is executed when called by the previous rule to copy
        -- the Association and its ownedEnd and memberEnd to the target model
unique lazy rule Association{
        from
                s : UML!Association
        to
                t : UML!Association(name<-s.name,ownedEnd<-s.ownedEnd,
                        memberEnd <- s.memberEnd) }
```

Properties related to a class (attributes), generalizations, and operations are elements contained into a class, so according to the UML containment hierarchies they are selected whenever their container is selected. The use case diagram is generated similarly to the class diagram (see [Tawhid & Petriu, 2011a] for more details).

After generating a specific product model, the generic performance specifications annotating it need to be bound to concrete values. The transformation collects from the generated UML product model all the generic performance parameters and all the PC-features from the PC-feature model, and associates each PC-feature to its corresponding model element(s). For instance, the PC-feature *dataCompression* is associated to a model element of type *message,* since it has an impact on the message size and the communication overheads. The transformation generates spreadsheets containing all the attributes that need binding, as seen in the following example.

```
        -- Rule Message2Row collects all the generic tagged values of the
        -- stereotypes «PaStep» or «PaCommStep» that annotate model element of
        -- type message and transforms them to a row in a table
rule Message2Row {
        from
                s : UML!Message (s.hasStereotype('PaStep')or
                                        s.hasStereotype('PaCommStep'))
        using {hostDemand_name : Sequence(String) = Sequence {
                'Message', s.name, 'PaStep', 'hostDemand',
                s.getAttrValue('PaStep','hostDemand').first()};
         msgSize_name : Sequence(String) = Sequence {
                'Message', s.name, 'PaCommStep', 'msgSize',
                s.getAttrValue('PaCommStep','msgSize').first()};}      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)}
```

Element Type	Element Name	Stereotype Name	Attribute Name	PC-Feature Group Name	PC-Feature Name	Guideline for Value	Generic Parameter	Concrete Value
Message	requisitionRequest	PaStep	hostDemand	application-annotation			$ReqSD (ms)	
		PaCommStep	msgSize	«exactly-one-of feature» dataCompression	compressed	msgSize reduced 10% ...30%		
					uncompressed	No effect		
							$MReq (KB)	
			commTxOvh	«exactly-one-of feature» secureCommunicatio	unsecured	No effect		
					secured			
				«exactly-one-of feature» secured	SSL Protocol			
					TLS Protocol			
				«exactly-one-of feature» securityLevel	lowSecurity	commTxOvh increased 5% ... 25%		
					mediumSecurity	commTxOvh increased 10% ... 35%		
					highSecurity	commTxOvh increased 15% ... 45%		
				«exactly-one-of feature» dataCompression	compressed	commTxOvh increased 2% ...5%		
					uncompressed	No effect		
							$MReqS(ms/KB)	
			commRcvOvh	«exactly-one-of feature» secureCommunicatio	unsecured	No effect		
					secured			
				«exactly-one-of feature» secured	SSL Protocol			
					TLS Protocol			
				«exactly-one-of feature» securityLevel	lowSecurity	commRcvOvh increased 5% ... 25%		
					mediumSecurity	commRcvOvh increased 10% ... 35%		
					highSecurity	commRcvOvh increased 15% ... 45%		
				«exactly-one-of feature» dataCompression	compressed	commRcvOvh increased 2% ...5%		
					uncompressed	No effect		
							$MReqRcv(ms/KB)	

Fig. 7. Part of the generated Spreadsheet *for the scenario* Create Requisition

A part of the generated spreadsheet for the scenario *Create Requisition* is shown in Fig. 7. For instance, the PC-feature *dataCompression* is mapped to message size through the MARTE attribute *msgSize* annotating a model element of type *message*. The column titled *Concrete Value* is designated for the user to enter the concrete value for each corresponding generic parameter, while the column *Guideline for Value* provides a typical range of values to guide the user. 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 SPL annotation variables.

Another kind of mapping that takes place is that of the generic processing nodes from the SPL deployment diagram to actual nodes for a specific product. Each lifeline in the sequence diagrams is stereotyped as «PaRunTInstance», providing an explicit connection at the annotation level between a role in a behaviour definition (a lifeline) and the corresponding runtime instance of an active object (process or thread), whose tag {host=$CustNode} indicates the physical node from the deployment diagram on which the instance is running. Thus, this tag needs to be bound to a concrete node for the product. The generated product model has either a centralized or distributed deployment diagram with maximum numbers of processors. The transformation collects all these processors' name and associates a list of these processors to each lifeline in the spreadsheets. The user will indicate a specific processor from this list.

After the user enters concrete values for all the generic performance parameters and selects an actual processor for each lifeline role provided in the spreadsheets, the second part of the model transformation takes as input these spreadsheets along with its corresponding product model, and binds all the generic MARTE tagged values in the product to the specific values provided in the spreadsheets. The outcome of this part of the transformation is a specific product model with concrete performance annotations for a specific PC-feature configuration, which can be further transformed automatically into a performance model.

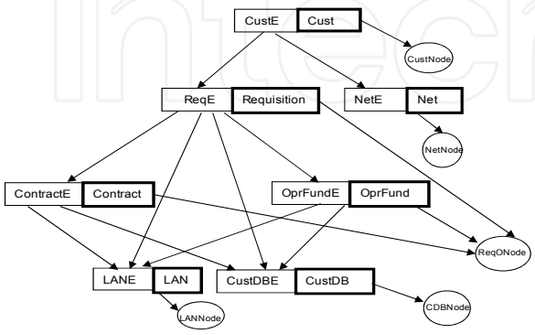

Fig. 8. Centralized LQN model

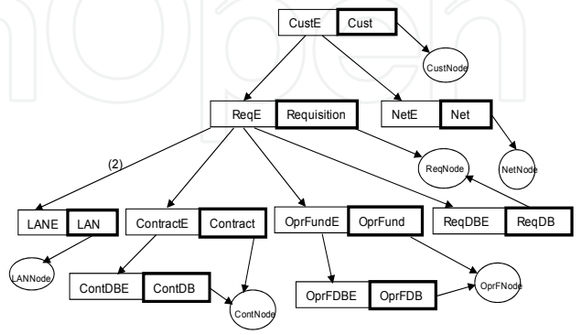

Fig. 9. Distributed LQN model

5. Performance analysis

As mentioned before, the derivation of a performance model from a SPL model requires two model transformations. The first transformation from the annotated SPL model to a product model with concrete performance annotations while the second one takes the target model of a concrete product and transforms it into a LQN performance model using the PUMA transformation approach [Woodside et al., 2005]. This section presents an example of an LQN performance model for the scenario *Create Requisition* shown in Fig.5 of a specific B2B system runs on two different architectures (centralized and distributed). Some performance analysis experiments conducted with the LQN models obtained for a concrete B2B system with a given PC-feature configuration is presented as well.

5.1 Performance model

The LQN model [Xu et al., 2003] is an extension of the well-known Queueing Network model developed for modelling software systems, which able to represent nested services. A software server often requires services from other servers in order to fulfil the requests of its own clients. An LQN model consists of a set of tasks that offer services represented by entries. The entries of a task may send requests to entries of other tasks. Software components are mapped to tasks while hardware devices mapped to hosts. Graphically, the software tasks are depicted as thick rectangles and the entries with attached thin rectangles. The hardware devices are represented as ellipses. LQN is used to model several types of system behaviour and inter-process communication style.

After obtaining the target model of a concrete product, it will be transformed into a LQN performance model using the PUMA transformation [Woodside et al., 2005]. The key performance scenario *Create Requisition* is transformed into two LQN models shown in Fig.8 and Fig.9 to represent the two different architectures; centralized and distributed,

respectively. In the centralized architecture, all customer database is allocated to the node *CDBNode* while, in the distributed architecture, the customer information is distributed over the three nodes *ReqNode*, *ContNode*, and *OprFNode*.

5.2 Performance results

Web-based applications, such as an e-commerce system that contains sensitive data and has many customers, require securing the data transmitted over certain communication channels. However, adding security may include a performance price. System designers need to make choices between different security levels and to make security/performance trade-offs. At the same time, it is important where the data is located in order to fulfill performance and security requirements. This location problem is examined in two different architectures: 1) distributed and 2) centralized. In the centralized architecture, all customer data is contained in one database. The centralized architecture has the advantage that updating and maintaining the data consistency is easier, but has the disadvantage of becoming the system bottleneck for large system sizes (when both the number of customers and the amount of data go up). A distributed architecture represents a solution where several databases divide the data and the work among them. It has potential for faster response times and improved performance, but makes the updates and keeping data consistency more difficult.

In order to illustrate the impact on performance of a secure communication channel between the browser and the webServer, a performance analysis experiment based on LQN models derived for B2B systems with different security levels running on two different architectures (centralized and distributed) is presented.

When a B2B system is generated, a specific configuration has to be selected from the PC-feature model. The key performance scenario *Create Requisition* in Fig. 5 is transformed into the LQN models shown in Fig. 8 and Fig. 9 used for experiments. Two configurations were chosen. The first configuration is for the centralized architecture where the customer database is running on the node *CDBNode* while the roles *Requisition*, *Contracts*, and *OperationFund* are running on the same node *ReqONode* as shown in Fig. 8. Furthermore, this node is linked to the *CDBNode* through a Local Area Network (LAN) channel with 1.0 ms latency. The connection between the *CustNode* and the *ReqONode* is set up through DSL Internet channels with 100 ms latency. The data is transmitted uncompressed with an average message size of 377.6 KB. The *CustomerAccount* database accesses an external device (hard disk) with an average read/write time of 77.1 ms. The second configuration is for the distributed architecture where the roles *RequisitionDB*, *ContDB*, and *OpFundDB* are running on the different nodes *ReqNode*, *contNode*, and *optFNode,* respectively as shown in Fig. 9. These nodes are connected through a LAN channel with 1.0 ms latency.

All communication channels in the unsecure system include no security solution, while the secure system contains certain secure channels using the TLS protocol. TLS has two phases: the handshake phase is used by the browser and webServer to exchange secrets and to generate a confidential symmetric key that is used for data exchange during data transfer, the second phase of the protocol. The public key encryption in the handshake phase may use keys of different lengths; a longer key provides a higher level of security, but the performance overhead increases. The strength of the symmetric encryption key and message

digest algorithms used by technology to exchange data may also vary, using strong encryption and authentication algorithms providing higher security. These algorithms are computationally intensive and add different performance overheads to the system. We used the data provided in [Menasce et al., 2004] for performance attribute values, which were obtained from measurements for three levels of security: the handshake overhead is of 10.2ms, 23.8 ms, 48.0 ms, and the data transfer overhead per KB of data is of 0.104 ms, 0.268 ms, 0.609 ms. The fourth case is for an unsecure system.

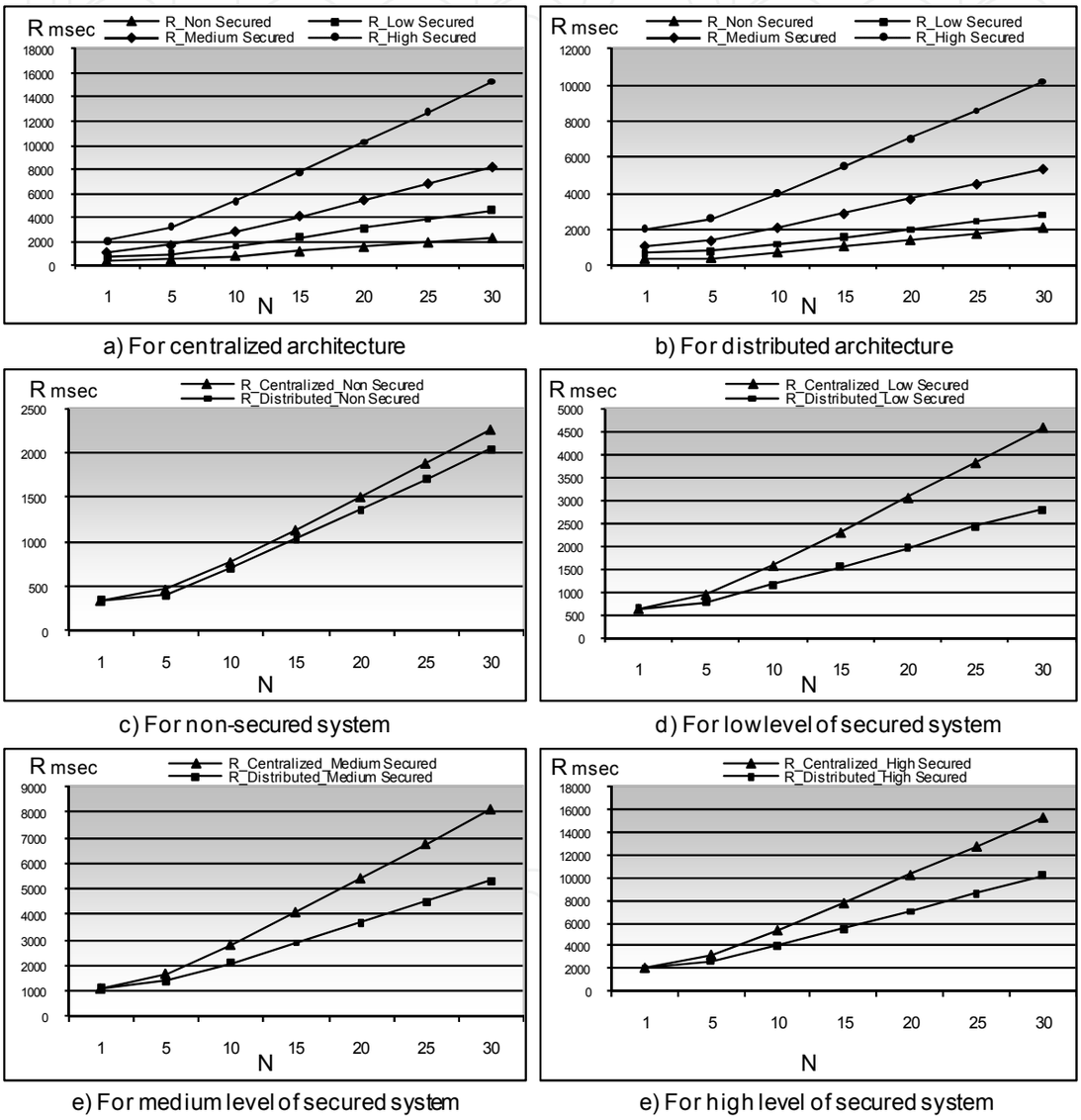

Fig. 10. Response time in function of the number of users

The LQN performance model is analyzed for different numbers of users with an existing solver [Franks, 2000]. Fig. 10a shows the response time of a user creating a requisition for different system choices (unsecure system and three security levels) running on centralized architecture, while Fig. 10b shows the same for a distributed architecture. Fig. 10c-f show the

response time in function of the number of simultaneous users executing the same scenario and running on the two different architectures for different levels of security.

The LQN results show that the secure system has a considerable effect on performance, as the response time for the secure system is much higher than for the unsecure system. As the number of users increases, the response time increases significantly due to the competition for resources. The *dataStorage* feature which is centralized or distributed has also a significant effect on performance, as the response time for the centralized architecture is significantly higher than for the distributed architecture for all levels of security.

This brief example illustrates the potential for performance analysis in early development stages, by allowing developers to analyze trade-off between two non-functional requirements, performance and security, and to compare the impacts of different design alternatives on performance. In general, a quantitative performance model helps the analyst to verify whether a system has the capacity to meet its performance requirements. It also helps indentifying the performance "hot spots" (e.g., the resources that will saturate first) and provides guidance for design or configuration changes in order to solve or mitigate the problems.

6. Conclusions

In this chapter, we propose to integrate performance analysis in the early phases of SPL model-driven development process. The goal is to help developers to evaluate the system performance and to choose better design alternatives as early as possible, so that the systems being built will meet their performance requirements. We start with a multi-view UML model of core family assets representing the commonality and variability between different products, which we call the SPL model. We add another dimension to the SPL model, annotating it with generic performance specifications expressed in the standard UML profile MARTE. A first model transformation derives the UML model of a specific product with concrete MARTE performance annotations from the SPL model. A second transformation generates a Layered Queueing Network performance model for the given product by applying an existing transformation approach named PUMA, developed in previous work. To the best of our knowledge, our research is the first to tackle the problem of generating a performance model for a specific product out of the SPL model. The main research challenges are rooted in the fact that a SPL model does not represent a uniquely defined system that could be implemented, run and measured as a whole, so we cannot talk about analyzing the SPL performance. A SPL model is instead a collection of core, generic asset models, which are building blocks for many different products with all kind of options and alternatives. Hence, we need to derive first a given product model with concrete performance-related details, and then we can consider transforming it into a performance model that can be used for performance analysis.

An important factor that distinguishes the SPL development from traditional software systems is variability modeling, a means of expressing the criteria that differentiate between SPL members. Different approaches for variability modeling have been proposed in literature, based on different concepts such as: features, variation points and variants, use case diagrams, or choices and decisions. Our approach employs the feature model to

represent variability between the family members, but we have also extended the use of features to express variability in performance completions.

We have considered developing a user-friendly approach. First, a mapping technique for explicit and implicit mapping of features to model elements is proposed, which aims to reduce the clutter of variability specifications in the SPL design model. Secondly, dealing manually with a huge number of generic performance annotations, by asking the developer to inspect every diagram in the model to extract these annotations in order to provide concrete binding values is an error-prone process. In this research, we automate the process of collecting all the generic parameters from the annotated UML model and present them in a user-friendly format to the user.

7. Acknowledgment

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

8. References

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

Bartholdt, J., Medak, M. & Oberhauser, R. (2009). Integrating Quality Modeling with Feature Modeling in Software Product Lines, *Proceedings of the 4th International Conference on Software Engineering Advances* (ICSEA2009), pp.365-370, 2009.

Belategi, L., Sagardui, G. & Etxeberria, L. (2010). Variability Management in Embedded Product Line Analysis, *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle* (VALID'10), pp. 69-74, Nice, France, 2010.

Belategi, L., Sagardui, G. & Etxeberria, L. (2010). MARTE mechanisms to model variability when analyzing embedded software product Lines, *Proceedings of the 14th International Conference on Software Product Line* (SPLC'10), pp.466-470, 2010.

Belategi, L., Sagardui, G. & Etxeberria, L. (2011). Model based analysis process for embedded software product lines, *Proceedings of the 2011 International Conference on Software and Systems Process* (ICSSP '11), 2011.

Botterweck, G., Lee, K. & Thiel, S. (2009). Automating Product Derivation in Software Product Line Engineering, *Proceedings of Software Engineering 2009* (SE09), pp 177-182, Kaiserslautern, Germany, 2009.

Clements, P. C. & Northrop, L. M. (2001). *Software Product Lines: Practices and Products*, Addison Wesley.

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

Czarnecki, K. & Antkiewicz, M. (2005). Mapping Features to Models: A Template Approach Based on Superimposed Variants, *Proceedings of the 4th international conference on Generative Programming and Component Engineering* (GPCE), LNCS vol. 3676, pp. 422–437, Springer, 2005.

Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau S. & Pietroszek, K. (2005). Model-Driven Software Product Lines, *Proceedings of the Object Oriented Programming Systems Languages and Applications conference*, OOPSLA, San Diego, California, 2005.

Czarnecki, K., Helsen, S. & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization, Software Process Improvement and Practice, pp. 7–29, 2005.

Franks, G. (2000). Performance Analysis of Distributed Server Systems, Report OCIEE-00-01, PhD. thesis, Carleton University, 2000.

Gamez, N. and Fuentes, L. (2011). Software Product Line Evolution with Cardinality-based Feature Models, *Proceedings of the 12th International conference on Software reuse* (ICSR 2011), pp. 102-118, 2011.

Gomaa, H. (2005). *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*, Addison-Wesley Object Technology Series, July 2005.

Groher, I. & Voelter, M. (2009). Aspect-Oriented Model-Driven Software Product Line Engineering, *Transactions on Aspect-Oriented Software Development* (AOSD) VI, LNCS, pp. 111-152, 2009.

Happe, J., Becker, S., Rathfelder, C., Friedrich, H. & Reussner, R. H. (2010). Parametric performance completions for model-driven performance prediction, Performance Evaluation, Volume 67 , Issue 8, pp. 694-716, 2010.

Heidenreich, F. &Wende, C. (2007). Bridging the Gap between Features and Models, *Proceedings of the 2nd Workshop on Aspect-Oriented Product Line Engineering* (AOPLE07) co-located with the 6th International Conference on Generative Programming and Component Engineering (GPCE¨07), 2007.

Heidenreich, F., Kopcsek, J. & Wende, C. (2008). FeatureMapper: Mapping Features to Models, *Proceedings of the 30th International Conference on Software Engineering* (ICSE08), pp. 943-944, New York, NY, USA, 2008.

Istoan, P., Biri, N. & Klein, J. (2011). Issues in Model-Driven Behavioural Product Derivation, *Proceedings of 5th International Workshop on Variability Modelling of Software-intensive Systems* (Vamos 2011), ACM, p. 69-78, Namur, Belgium, 2011.

Menasce, D., Almeida, V. & Dowdy, L. (2004*). Performance by Design: Computer Capacity Planning by Example*, Prentice Hall PTR, Upper Saddle River, NJ 07458, 2004.

Object Management Group, "UML: Super-structure", Version 2.1.2, OMG document formal/2007-11-02, 2007.

Object Management Group (2011). UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE),Version 1.1, OMG document formal/2011-06-02, 2011.

Raatikainen, M., Niemelä, E., Myllärniemi, V. & Männistö, T. (2008). Svamp – An Integrated Approach for Modeling Functional and Quality Variability, *Proceedings of the 2nd International Workshop on Variability Modeling of Software-intensive Systems* (VaMoS), 2008.

Stoiber, R. & Glinz, M. (2009). Modeling and Managing Tacit Product Line Requirements Knowledge, *Proceedings of the 2nd Interna-tional Workshop on Managing Requirements Knowledge* (MaRK09), at RE'09, Atlanta, USA, 2009.

Street, J. & Gomaa, H. (2006). An Approach to Performance Modeling of Software Product Lines, *Workshop on Modeling and Analysis of Real-Time and Embedded Systems*, Genova, Italy, October 2006.

Tawhid, R. & Petriu, D.C. (2008). Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line Model, *Proceedings of the 2008 ACM Int. Work-shop on Software Performance* (WOSP08), pp. 91-102, 2008.

Tawhid, R. & Petriu, D.C. (2008). Integrating Performance Analysis in the Model Driven Development of Software Product Lines, *Proceedings of MODELS* 2008, LNCS Vol. 5301, pp. 490-504, 2008.

Tawhid, R. & Petriu, D.C. (2011). Product Model Derivation by Model Transformation in Soltware Product Lines, *Proc. 2nd IEEE Workshop on Model-based Engineering for Real-Time Embedded Systems* (MoBE-RTES 2011), Newport Beach, CA, USA, 2011.

Tawhid, R. & Petriu, D.C. (2011). Automatic Derivation of a Product Performance Model from a Software Product Line Model, *Proceedings of the 15th International Conference on Software Product Line* (SPLC'11), Munich, Germany, 2011.

Woodside, M., Petriu, D. C. & Siddiqui, K. H. (2002). Performance-related Completions for Software Specifications, *Proceedings of the 22rd International Conference on Software Engineering*, ICSE 2002, pp. 22-32, Orlando, Florida, USA, 2002.

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

Xu, J., Woodside, C.M. & Petriu D.C. (2003). Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time, *TOOLS'2003, (P.Kemper and W.Sanders, eds.)* Springer LNCS Vol. 2794, pp.291-307, 2003.

Software Product Line - Advanced Topic

Edited by Dr Abdelrahman Elfaki

ISBN 978-953-51-0436-0

Hard cover, 122 pages

Publisher InTech

Published online 04, April, 2012

Published in print edition April, 2012

The Software Product Line (SPL) is an emerging methodology for developing software products. Currently, there are two hot issues in the SPL: modelling and the analysis of the SPL. Variability modelling techniques have been developed to assist engineers in dealing with the complications of variability management. The principal goal of modelling variability techniques is to configure a successful software product by managing variability in domain-engineering. In other words, a good method for modelling variability is a prerequisite for a successful SPL. On the other hand, analysis of the SPL aids the extraction of useful information from the SPL and provides a control and planning strategy mechanism for engineers or experts. In addition, the analysis of the SPL provides a clear view for users. Moreover, it ensures the accuracy of the SPL. This book presents new techniques for modelling and new methods for SPL analysis.

InTech Europe

University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen