Integrating Performance Analysis in the Model Driven Development of Software Product Lines

Rasha Tawhid[1] and Dorina Petriu [2]

[1]School of Computer Science, Carleton University, Ottawa, Canada,
rtawhid@connect.carleton.ca
[2]Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada,
petriu@sce.carleton.ca

Abstract. The paper proposes to integrate performance analysis in the early phases of the model-driven development process for Software Product Lines (SPL). We start by adding generic performance annotations to the UML model representing the set of core reusable SPL assets. The annotations are generic and use the MARTE Profile recently adopted by OMG. A first model transformation realized in the Atlas Transformation Language (ATL), which is the focus of this paper, 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. The proposed technique is illustrated with an e-commerce case study that models the commonality and variability in both structural and behavioural SPL views. A product is derived and the performance of two design alternatives is compared.

Keywords: Software Product Line, Performance Analysis, Model to model Transformation, UML, MARTE, ATL

1 Introduction

Software Product Line (SPL) engineering aims at improving productivity and reducing development time, effort, cost, and complexity by gathering the analysis, design and implementation activities of a family of systems. It is based on the reuse of core assets instead of working from scratch. An important challenge in the context of SPL approach is to model and manage variability between products and to support the derivation of specific products from the family assets.

Model-driven development (MDD) improves software development by capturing the key features of a system in models which are developed and refined as the system is created [18]. Many existing works have investigated ways of applying MDD to SPL development, with the goal of generating code for given products from the SPL model. In this paper, we propose to add another dimension to the model-driven development of SPL, by generating a performance model for a given product from the SPL model, in the early development phases. Early performance analysis allows

developers to gain insight on the performance trouble spots for different design alternatives under different workload conditions. 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.

Evaluating non-functional properties from UML models is possible by adding first additional information specific to the property to be evaluated, and then transforming the annotated UML model into a formal model which can be analyzed with known analysis techniques and tools [20]. Examples of formal models used for performance analysis are queueing networks, Petri nets, stochastic process algebras, etc. [3]. The "UML Performance Profile for Schedulability, Performance and Time" (SPT) standardized by OMG or its recent replacement, the "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)" [14] define quantitative performance annotations (such as resource demands made by different software execution steps, visit ratios, performance requirements, etc.) to be added to a given UML model, particularly to the architecture, behaviour and deployment views.

In literature there are many publications investigating the application of MDD to SPL, as discussed in section 2. Also, there is a body of work on software performance engineering aiming to build performance models from software specifications, which can be further used for early performance analysis [3]. However, to the best of our knowledge, our research is the first to propose an approach for integrating quantitative performance analysis in the early phases of UML-based model-driven development of SPL. The main research challenge stems from the mismatch between what a SPL model and a performance model represent. A SPL model is a collection of core "generic" asset models, which are building blocks for many different products with all kind of options and alternatives, while a performance model is an instance-based representation of a runtime system, focusing on how the system is using all kind of resources and how the competition for resources impacts the system performance (response time, throughput, utilization, etc.). Hence, a first research challenge is SPL-related: automating the derivation of a UML product model that contains all the views necessary for performance analysis (i.e., software architecture, key-performance scenarios and deployment) from a SPL model. A second research challenge is performance-related, due to the fact that a new dimension is added to the model transformation when dealing with performance annotations. The SPL model should have reusable generic parametric performance annotations, which will be bound to concrete values when generating the model of a specific product.

The approach proposed in the paper, illustrated in Fig.1, requires two transformations. The starting point is a UML model of a SPL with generic performance annotations, which uses two separate profiles: a "product line" (PL) profile similar to [10] for specifying the commonality and variability between products, and MARTE for performance annotations [14]. The first model-to-model transformation takes as input the SPL source model and a set of binding directives specifying the mapping between the generic and concrete performance annotations (in XML format). The *target model* of this transformation is a UML+MARTE model of a product, where the variability expressed in the SPL model has been analyzed and bound to a specific product, and the generic performance annotations have been bound to concrete values. This transformation uses the Atlas Transformation Language (ATL) [1] which is based on the Eclipse Modeling Framework (EMF).

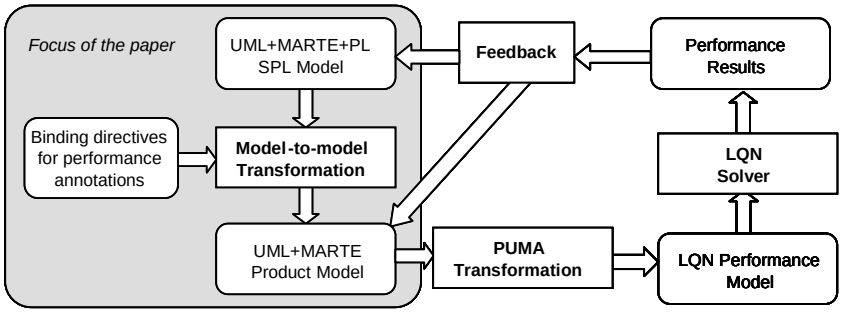

Fig. 1. Approach for integrating performance analysis with MDD of SPL

The second transformation takes as input the outcome of the first transformation and derives a Layered Queueing Network (LQN) performance model for the specific product, by using the PUMA transformation approach that has been previously developed in our research group [20][21]. After the performance model for a product was generated, it can be analyzed with existing LQN solvers and feedback regarding its performance properties will be given to the software development team. The focus of this paper is on the first transformation shown in the shaded area.

The proposed technique is illustrated with an e-commerce case study, which models the commonality and variability in both structural and behavioural views similar to the Product Line UML-based Software Engineering (PLUS) method [10]. (The differences between our approach and PLUS are discussed in section 2). The e-commerce web service is a high performance distributed application where several architectural questions arise. One of them is the location of data storage (centralized or distributed). We consider that this architectural decision is a *quality feature* (as opposed to a *functional* one) because it impacts non-functional requirements or concerns, such as performance, availability, security, reliability, etc. We propose to take into account quality features from the early phases of the SPL development process, and to represent their relationships with the functional features in the feature dependency diagram. Integrating performance analysis into SPL in the early development phases allows assessing the impact of different choices for the quality and functional features on system performance.

The paper is organized as follows: section 2 discusses related research, section 3 presents the transformation algorithm for product derivation, section 4 analyzes the performance effects of a case-study and discusses different design alternatives and section 5 presents the conclusions and future work.

2 Related Research

In this section, we discuss briefly related research on two topics: model-driven development of SPL, and performance analysis of UML models annotated with performance information (with SPT or MARTE) in early development phases.

A lot of work has been done in the area of integrating MDD into SPL to achieve the benefits of the two paradigms. The ultimate MDD objective in most of the cases is to generate code for a product from the SPL model; in some cases, however, a product model is also obtained.

In [5] is presented an approach for deriving the architecture of a product by selectively copying elements from the SPL architecture based on a product-specific feature configuration. Another approach in [13] describes the general architecture of the family and the variation configuration model. Variation points are specified in the SPL class view by using the stereotype <<vp>> implying a variation that needs to be resolved at configuration time. Based on these variation points, a graphical decision model is generated to configure the product architecture.

A domain-specific language called Model Template Transformation Language (MTTL) for specifying the transformations of model templates based on feature models is described in [2]. First, the product line model is developed by creating a feature model and a model template. Then, according to the selected feature, a feature configuration is created and an Atomic Transformation Code (ATC) is executed for specializing the model template.

In [18] aspect-oriented techniques are used to manage variability in SPL. In the problem domain, SPL is modeled using a DSL, where each variation needs to be configured with various options. In the solution domain, a component-based architecture is built starting with the minimal core and selectively adding additional parts by weaving aspects.

In [8] a feature-based model template is introduced, which consists of annotated models implementing the features. A template instance for a given feature configuration can be produced automatically. In [9] a generic two-phased product derivation process is presented. In the initial phase, a first configuration is created from the product family assets and modified in a number of subsequent iterations until the product satisfies all its requirements. In [15] the variability of a product line is modeled and realized by higher-order transformations using the MOFScript language. Generative programming is combined with aspects to represent the variability in SPL. A process to obtain a use case model for a specific application based on a feature configuration model is described in [6].

The work by Jézéquel et al. addresses product derivation at structural and behavioural levels [22][23]. An approach for deriving a product model based on a creational design pattern is proposed in [22]. A model derivation technique for static and behaviour views is proposed in [23]. The static derivation starts from a SPL class diagram and generates the product class diagram based on a decision model. An algebraic approach is proposed to derive statecharts for a specific product from the sequence diagrams of the product line, by transforming product scenarios given as a reference expression for SD into a composition of statecharts.

Another group addressing UML-based product derivation is Gomaa's group. In [10] a method called Product Line UML-based Software Engineering (PLUS) for modeling explicitly the commonality and variability in a SPL is presented. One of the few papers proposing tool support for multiple-view SPL models stored in a repository is [11]. Automated support for product derivation from the product line repository is also proposed in [11]. A modeling approach for dynamic reconfiguration of pattern-based software architectures is presented in [12].

It is worth mentioning that feature modeling is essential in SPL, yet the concept of "feature" is not a first-class model element in UML. Thus, we cannot use the traditional feature diagram in UML models. 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 work is based on Gomaa's group work, especially on PLUS [10] for the following reasons: it is a well developed method applied to real-time systems, is concerned with the behaviour view needed for performance analysis, and uses a profile for extending UML with SPL concepts (which we use in this paper under the name "PL profile"). However, our approach has the following differences from PLUS: a) we deal with MARTE performance annotations both in the source and target models; b) we introduced the concept of "quality feature" described in section 3; c) we use sequence diagrams for behaviour representation instead of collaboration (communication) diagrams, taking advantage of their enhanced modeling power; d) we use deployment diagrams, also important for performance analysis; e) we modified slightly the PL stereotypes and tags in order to represent quality features.

Software Performance Engineering (SPE) is a methodology introduced in [17] that promotes the integration of performance analysis into the software development process from the early stages and continuing throughout the whole software life cycle. Since the introduction of SPE, there has been a significant effort to integrate performance analysis into the software development process by using different performance modeling paradigms: queueing networks, Petri nets, stochastic process algebras, simulation, etc. [3]. The performance modeling formalism used in this paper is the Layered Queueing Model (LQN) [19]. A good survey of transformations of software models into different performance models is given in [3]. Examples of such transformations are from UML to Layered Queueing Networks in [16], to Stochastic Petri Nets in [4], and to Stochastic Process Algebra in [7]. In this work we are using the transformation framework PUMA described in [20][21], which converts an annotated UML model of a concrete system into different performance models (Layered Queueing Networks, Queueing networks, Petri Nets). Usually, the interpretation of the performance model results is done by a performance analyst, who understands the formal performance model. Current research is being done to diagnose the performance problems automatically (by following a set of rules similar to the experts) and to suggest advice for improvement in terms that the software developers can easily understand.

3 Product Model Derivation

There are two major processes in SPL engineering: a) *domain engineering* for analyzing the commonality and variability between members of the product line and establishing reusable SPL models, and b) *application engineering* for deriving an individual product that is a SPL member from reusable SPL models.

In UML-based domain engineering, we represent SPL features as use case packages and feature dependency as stereotyped class diagram, describing all feature combinations possible with this SPL; also, we use a SPL class diagram, sequence

diagrams, deployment diagram describing the overall views for this product line. To illustrate the proposed derivation process, we use an e-commerce case study similar to [10], with some modifications. The e-commerce SPL is a web-based product line that handles business-to-business (B2B) as well as business-to-consumer (B2C) systems. For example 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 for fund availability. Optionally, a supplier may create a purchase order requesting new inventory supplies from the wholesaler.

3.1 Source Model

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

The use case diagram for the e-commerce SPL is given in Fig. 2. 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. The use cases are grouped by type in packages (not shown here due to space limitations). Each package corresponds to a feature bearing the same name as the package. For instance, the use cases from Fig. 2 can be grouped as follows: the kernel use cases in a package "E-Commerce Kernel", the optional use cases in "Purchase Order", the alternative use cases for B2C in "Home Customer", and the alternative cases for B2B in "Business Customer".

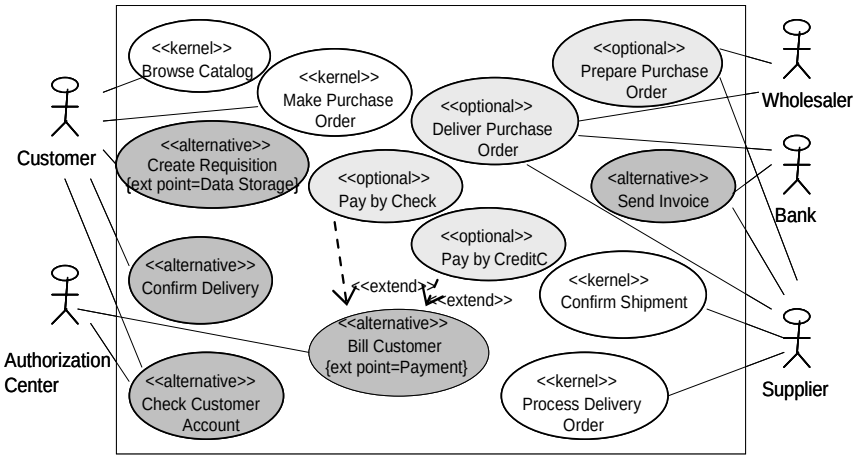

Fig. 2. Use case model of e-commerce SPL

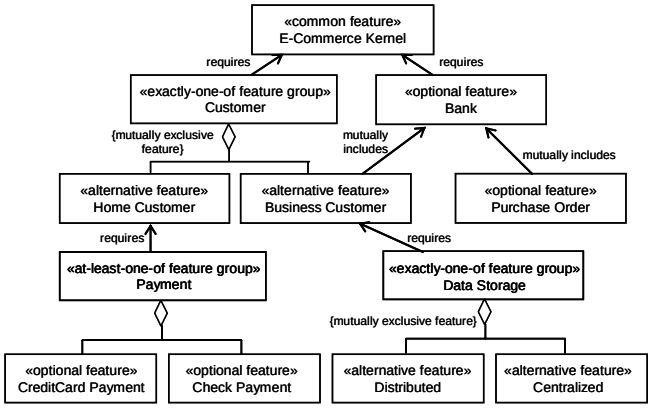

Fig. 3. Feature dependency in the e-commerce SPL

Fig.3 illustrates the feature dependency represented as a class diagram with PL stereotypes, describing the way features can be combined within this SPL. For instance, the two alternative features "Business Customer" and "Home Customer" are mutually exclusive, hence they are grouped into an *exactly-one-of* feature group called "Customer". Beside the functional features, we add to the diagram so-called *quality features* characterizing design decisions that have impact on the non-functional requirements or concerns. For example the architectural decision related to the location of data storage (Centralized or Distributed) affects performance, and is represented in the diagram by two mutually exclusive quality features.

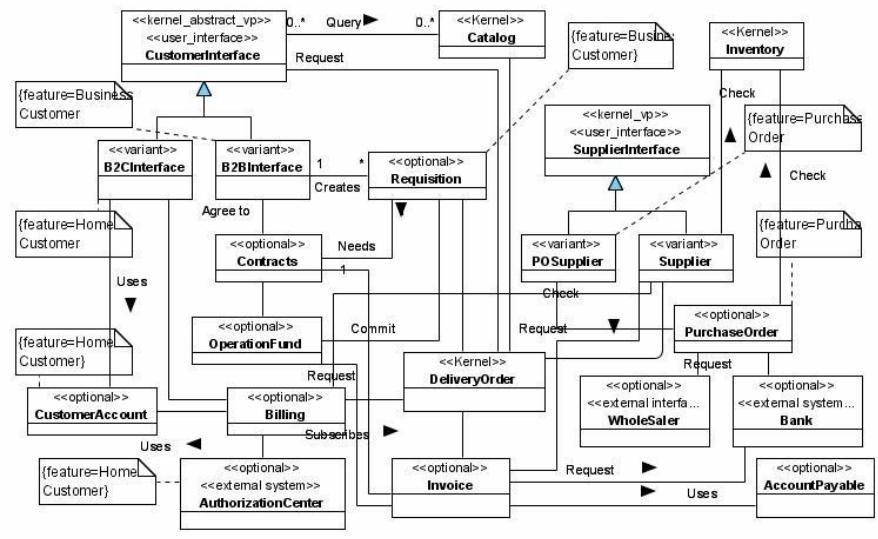

Fig. 4. Class diagram of e-commerce SPL

The class diagram for the e-commerce SPL in Fig. 4 shows that classes are stereotyped in three categories: kernel, variant or optional. The stereotypes for *variant* and *optional* classes have a tag indicating the feature(s) requiring that class. A generalization/specialization hierarchy is used to model classes that behave differently in B2B and B2C systems (such as CustomerInterface and Supplier Interface).

For each scenario of each use case, at least a sequence diagram is created. Fig. 5 illustrates the scenario CreateRequisition, one of the 15 scenarios created for the case study. The sequence diagram itself is stereotyped as «GaPerformanceContext», indicating that this diagram is to be considered for performance analysis. This stereotype may have a set of parameters defining global properties of this analysis context, held by its tag contextParams. (In this example, the context has no global parameters).

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 run time instantiation of a process (active object). The tag {instance=Requisition} indicates which run-time instance of a process executes the lifeline role.

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 ExecutionOccurence (represented as thin rectangle on the lifeline) or to the message that triggers it. For instance, in Fig.5, the message ContractQuery is stereotyped as an execution step:

«PaStep» {hostDemand=($ContD, ms)}

where the tag hostDemand indicates the execution time required by the step, which is given by the variable $ContD in time units of milliseconds. (In MARTE, the variables start with '$'). Note that using a variable for the execution time makes this a generic annotation that will be bound to a concrete value when deriving a given product. Both SPT and MARTE allow for variables and expression in annotations in order to raise the level of abstraction and make the annotations more reusable.

The same message ContractQuery is also stereotyped as a communication step:

«PaCommStep»{msgSize = ($ContQ,KB)}

where the message size is given by the variable $ContQ in KiloBytes.

The workload of a scenario is defined as a stream of events driving the system which can be open or closed. In our example:

«GaWorkloadEvent» {closed (population=$N2),(extDelay=$Z2)}

the workload is closed with a number of users $N1 and user think time for a user $Z2.

The alt fragments whose choices are based on the value of the quality feature DataStorage (which is represented as a scenario variation point) are stereotyped with the PL stereotype <<extension point>>{extension=DataStorage}.

Finally, the deployment diagram for the SPL is created assuming maximum distribution. It contains all the possible artifacts contained in all the products, even artifacts corresponding to optional or variant classes. Maximum distribution means providing the largest number of processors that might ever be used for any product of the SPL, it doesn't mean providing a processor for every 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.

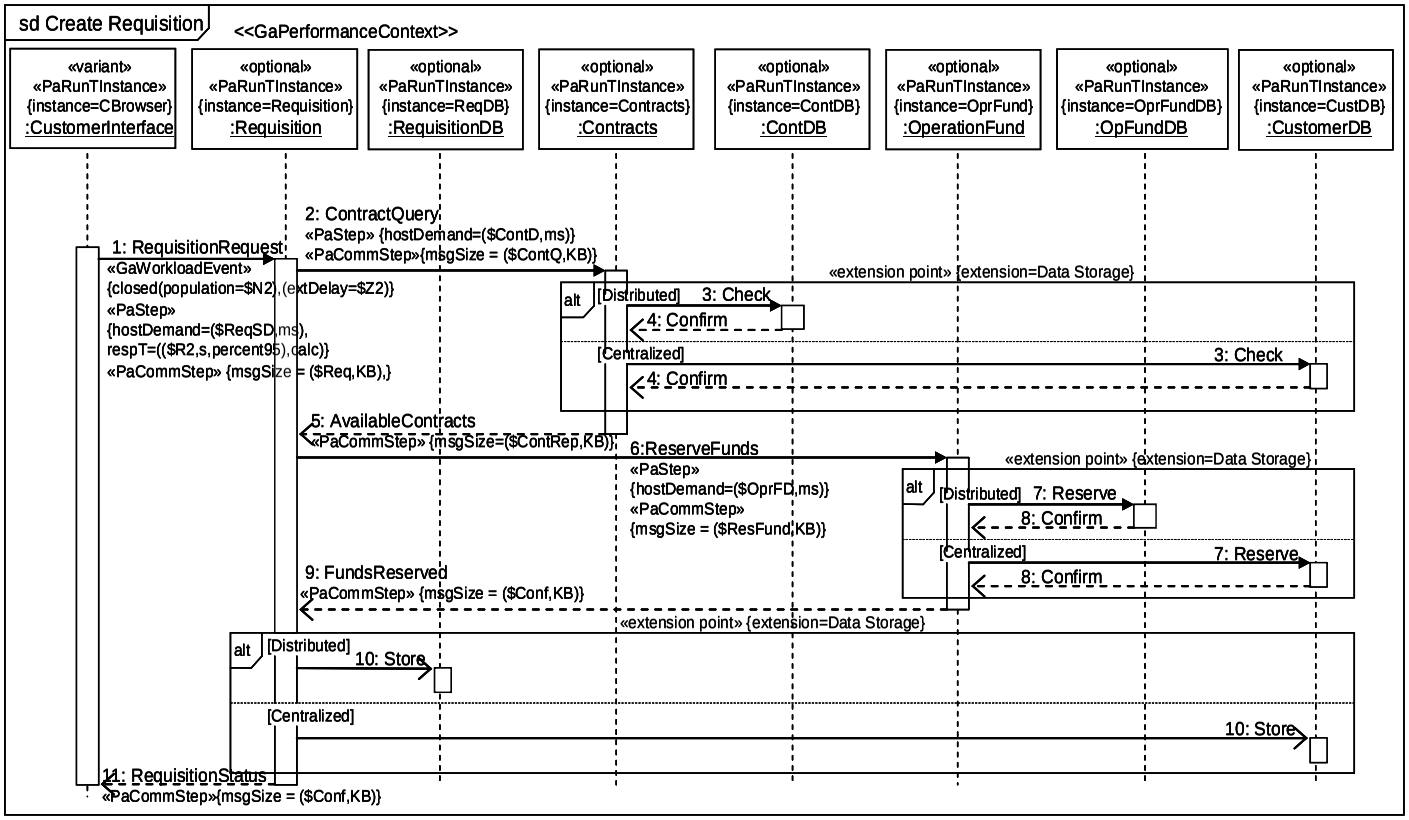

Fig. 5. SPL scenario CreateRequisition

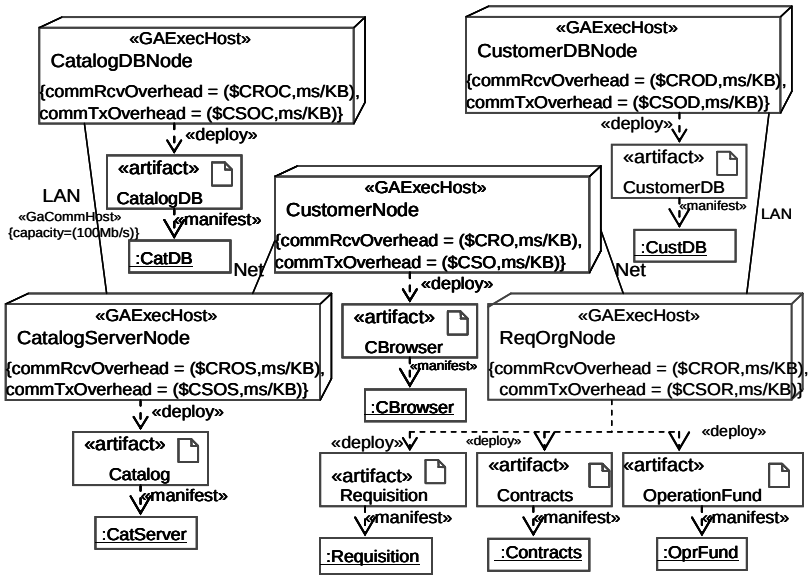

Fig. 6. Part of the SPL deployment diagram for the centralized architecture

Fig. 6 shows a part of the deployment diagram for the centralized architecture. Each processing node in the deployment diagram is stereotyped as an execution host with stereotype «GaExecHost». The node may be stereotyped with communication overheads. The attributes commRcvOverhead and commTxOverhead are the host demand overheads for receiving messages and sending messages, respectively

3.2 Target Model

The target model represents a product, so it does not contain any PL profile extensions because the variability has been resolved. However, the product model contains performance annotations that have been bound to concrete values, as indicated by the user. The product model consists of a use case view, class diagram, sequence diagram for each scenario and deployment diagram.

Table 1 shows the mapping of the annotation variables to concrete values for two scenarios used in section 4, BrowseCatalog and CreateRequisition. Choosing the values to be assigned to the performance parameters from the SPL model is not a simple problem. In general, it is difficult to estimate quantitative resource demands for each activity in the design phase, when an implementation does not exist and cannot be measured yet. Several approaches are used by the 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 estimate can be replaced with measured values for the critical parts, increasing the model accuracy.

Table 1. Mapping of annotation variables to concrete values

generic parameters	concrete values	generic parameters	concrete values	generic parameters	concrete values
$N1	[10..1000]	$N2	0.1 * N1	$ContD (ms)	0.9
$Z1 (ms)	0	$Z2 (ms)	0	$ContQ (KB)	3
$R1 (s)	[0.7..67]	$R2 (s)	[0.9..56]	$oprFD (ms)	1.1
$CatSD (ms)	1	$ReqSD (ms)	1.1	$ContRep (KB)	2
$GetL (KB)	0.5	$Req (KB)	3	$Conf (KB)	0,5

Another kind of binding that takes place during the derivation of a specific product model from SPL is the binding of the generic roles associated to sequence diagram life-lines to the desired role for handling the chosen feature(s).

3.3 Model Transformation

This section presents briefly a prototype implementation for the derivation of a given product as a model transformation realized in the Atlas Transformation Language (ATL) [1]. The ATL transformation takes as inputs the source model described in section 3.1 as well as the PL profile and the MARTE profile and generates the target model for a product presented in section 3.2. The derivation process starts by selecting the features for the product we want to develop. The chosen features are checked against the feature dependency diagram from the source model to identify any inconsistencies between features. The steps of the proposed model transformation algorithm are presented in Fig. 7. Assume that the transformation is applied to the e-commerce case study to derive the business-to-business (B2B) model consisting of the use case, class, sequence and deployment diagrams from the e-commerce SPL model. Due to limited space, we discuss only the derivation of the product class diagram from the SPL class diagram (but the other diagrams are derived by following a similar approach of selectively adding only the elements required for the desired product). Since the SPL class diagram represents the union of all possible product class diagrams, the derivation can be done by selecting and copying the classes from the SPL class diagram to the product class diagram one by one. This derivation starts with the minimal core (in our case the kernel classes) and selectively adds additional classes based on the chosen features.

The ATL model transformation takes as inputs the SPL class diagram shown in Fig. 4. The source metamodel is the UML metamodel extended with the PL and MARTE profiles, and the target metamodel is the UML metamodel extended with MARTE only. The outcome of the transformation is the product model. From the SPL class diagram from Fig. 4, the kernel classes CustomerInterface, Catalog, SupplierInterface, Inventory, and DeliveryOrder are copied first into the product model, keeping the same name but removing the PL-related stereotypes. In the source model, each class is annotated with the feature that requires it. These annotations are represented as the values of the property in the stereotype for this class. The optional

BEGIN

1. Select the desired features for the product and check them for consistency.

2. Select use cases realizing the chosen features from the SPL use case diagram
   SWITCH
   CASE 1: Feature realized as use case package
   BEGINCASE
   Select all use cases in the package;
   ENDCASE
   CASE 2: Feature realized as "extend" or "include" relationships between use cases
   BEGINCASE
   Select the base use case and the included or extended use cases;
   ENDCASE
   CASE 3: Feature realized as a variation point within a scenario realizing the use case;
   BEGINCASE
   Select the respective use case;
   ENDCASE
   ENDSWITCH

3. Derive the product class diagram from the SPL class diagram.
   Select "kernel" classes;
   Select "optional" or "variant" classes corresponding to the chosen features;
   Select associations between selected classes;

4. **FOR** each scenario of the selected use cases
   Choose the corresponding sequence diagram;
   FOR each chosen sequence diagram
   Bind each generic role associated to a life-line to the desired role for handling the chosen feature;
   Bind the performance annotations to concrete values provided by the user;
   ENDFOR
   ENDFOR

5. Build the product deployment diagram from the SPL deployment
   Determine product artifacts;
   Bind generic processing nodes to actual ones;
   Bind generic performance annotations to concrete values;

END

Fig. 7. Steps of model transformation algorithm

and variant classes are selectively copied to the target model according to the value of the property of the optional and variant stereotypes. For instance in this case the variant classes tagged with BusinessCustomer will be selected; the B2BInterface and Supplier classes are copied into the product model. Similarly, the optional classes will be copied. Finally the associations between these classes will be copied, if both classes attached to the association ends have been already copied.

Here we show an example of an ATL helper, hasStereotype, which defines how an element can be retrieved according to the name of its stereotype. This helper is called by the rule KernelClass which copies the classes stereotyped as kernel from the source to the target model, by keeping the same name.

```
helper context UML!Element def:hasStereotype(stereotype:String) :Boolean
= self.getAppliedStereotypes()-> exists (c |
c.name.startsWith(stereotype));

rule KernelClass{
       from
              s : UML!Class (s.hasStereotype('kernel'))
       to
              t : UML!Class (
                     name <- s.name,
                     ownedAttribute <- s.ownedAttribute,
                     ownedOperation <- s.ownedOperation
                     )  }
```

The following helper, which retrieves the value of the property with the specified name in the specified stereotype for this element, is used by the rules that copy the optional and variant classes:

```
  helper context UML!Element def: getTaggedValue(stereotype : String,
         tag : String) : String =
         self.getValue(self.getAppliedStereotype(stereotype),tag);
```

4 Performance Analysis

After the target model of a concrete product is generated, it is further transformed into a LQN performance model using the PUMA transformation approach [20][21]. This section presents some performance analysis experiments conducted with the LQN models obtained for the B2B system for the centralized and distributed architectures.

Two key performance scenarios, BrowseCatalog and CreateRequisition, for the centralized B2B system are transformed together into the LQN models used for experiments. In the e-commerce application, 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. Fig. 6 shows the centralized design with only one node, the CustomerDB Node that stores the customer 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.

We solved the LQN models for different numbers of users and compared the two B2B systems, which differ only in the choice of the Data Storage feature: Centralized and Distributed. The effects of the two architecture choices on the response time R1 perceived by a user who is browsing the catalog, and the response time R2 of a user placing a requisition are compared in Fig.8. We assumed that the system is used concurrently by N1 users who browse the catalog and N2 who place a requisition, where N1 is increasing from 10 to 1000 and N2 from 1 to 100, respectively. The LQN results show that the Data Storage feature has a considerable effect on performance as shown in Fig. 8, as the response time for the centralized architecture is significantly higher than for the distributed architecture for both user types.

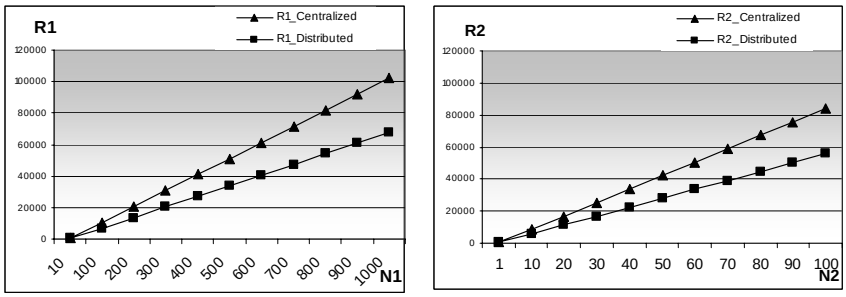

Fig.8. Response time in function of the number of users for the two different architectures

This brief example illustrates the potential for performance analysis in early development stages, by allowing the developers to compare the performance effects of different design alternatives.

5 Conclusions

The Software Product Lines development process takes advantage of the reusability of a set of core assets shared among the members of a family of products, instead of building each product from scratch. In this paper, we propose to integrate performance analysis in the UML-based model-driven development process for SPL by adding generic performance annotations to the SPL model and reusing them when deriving a specific product.

To the best of our knowledge, our research is the first to tackle this problem. The main research challenges are rooted in the fact that a SPL model does not represent a clearly defined system that could be implemented, run and measured, so we cannot talk about analyzing its 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 concrete product with a well-defined structure and behaviour, and then we can consider analyzing its performance. The challenges of the proposed research are both SPL-related and performance-related.

Regarding the derivation of a product model from an SPL model, we are planning to consider in the future aspect-oriented modeling techniques for weaving new structural and behavioural elements into a product model. Regarding the challenge of dealing with performance annotations, we will investigate whether MARTE has all the necessary features to allow for expressing parametric generic reusable annotations. We also plan to devise a more user-friendly way than XML for expressing binding directives that handle a large amount of data.

Acknowledgments

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).

References

1. Atlas Transformation Language (ATL), http://www.eclipse.org/m2m/atl.
2. Avila-García, O., García, A. E., Sánchez Rebull, E. V.: Using Software Product Lines to Manage Model Families in Model-Driven Engineering. In ACM symposium on Applied Computing, pp. 1006--1011, Seoul, Korea (2007).
3. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. In IEEE Transactions on Software Engineering, vol. 30, N.5, pp.295--310 (2004).

4. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable Petri net models. In 3rd International Workshop on Software and Performance (WOSP02), pp. 35--45, Rome (2002).
5. Botterweck, G., O'Brien, L., Theil, S.: Model-driven derivation of product architecture. In 22nd IEEE/ACM international conference on Automated software engineering, pp. 469--472, Atlanta, Georgia, USA (2007).
6. Braganca, A., Machado, R. J.: Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. In: 11th International Software Product Line Conference (SPLC), Kyoto, Japan (2007).
7. Cavenet, C., Gilmore, S., Hillston, J., Kloul, L., Stevens, P.: Analysing UML 2.0 activity diagrams in the software performance engineering process. In 4th International Workshop on Software and Performance (WOSP 2004), pp. 74--83, Redwood City, CA (2004).
8. Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau, S., Pietroszek, K.: Model-Driven Software Product Lines. In OOPSLA, San Diego, California (2005).
9. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: A case study. In Journal of Systems and Software, vol. 74, pp. 173--194 (2005).
10. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-based software Architectures. Addison-Wesley Object Technology Series (2005).
11. Gomaa, H., Shin, M. E.: Automated Software Product Line Engineering and Product Derivation. In 40th Hawaii International Conference on System Sciences (2007).
12. Gomaa, H., Hussein, M.: Model-Based Software Design and Adaptation. Int. Conference on Software Engineering for Adaptive and Self-Managing Systems, p. 7 (2007).
13. Haugen, O., MOller-Pedersen, B., Oldevik, J., Solberg, A.: An MDA-based framework for model-driven product derivation. In: M. H. Hamza, editor, Software Engineering and Applications, pp. 709--714. ACTA Press, Cambridge (2004).
14. Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded Systems, OMG Adopted Specification ptc/07-08-04 (2007).
15. Oldevik, J., Haugen, O.: Higher-Order Transformations for Product Lines. In: 11th Int. Software Product Line Conference (SPLC), pp. 243--254, Kyoto, Japan (2007).
16. Petriu, D.C., Shen, H.: Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications. In Comp. Performance Evaluation (T. Fields, P. Harrison, J. Bradley, U. Harder, Eds.) LNCS 2324, pp.159--177 (2002).
17. Smith, C.U., *Performance Engineering of Software Systems*, Addison Wesley, (1990).
18. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: 11th International Software Product Line Conference (SPLC), Kyoto, Japan (2007).
19. Woodside, C.M., Neilson, J.E., Petriu, D.C., Majundar, S.: The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. In IEEE Trans. on Computers, vol.44, Nb.1, pp. 20—34 (1995).
20. Woodside, C.M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by Unified Model Analysis (PUMA). In WOSP'05, Palma de Mallorca, Spain (2005).
21. Woodside, C.M., Petriu, D.C., Xu, J., Israr, T., Merseguer, J.: Methods and Tools for Performance by Unified Model Analysis (PUMA). Technical Report SCE-08-06, Carleton University, Systems and Computer Engineering, 35 pages (2008).
22. Ziadi, T., Jézéquel, J.M., Fondement, F.: Product line derivation with uml. In Software Variability Management Workshop, pp 94–-102, University of Groningen Department of Mathematics and Computing Science (2003).
23. Ziadi, T., Jézéquel, J.M.: Product Line Engineering with the UML: Deriving Products. In Software Product Lines, pp 557--586, Springer (2006).