

Tool Support for Software Variability Management and Product Derivation in Software Product Lines

Hassan Gomaa¹, Michael E. Shin²

¹ Dept. of Information and Software Engineering, George Mason University,
Fairfax, VA 22030-4444, USA
hgomaa@gmu.edu

² Dept. of Computer Science, Texas Tech University,
Lubbock, 79409-3104, USA
Michael.Shin@coe.ttu.edu

Abstract. Software variability management is a key challenge in developing software product lines and deriving products from the product line. In order to provide effective variability management and product derivation in software product lines, which is capable of being automated, certain fundamental building blocks are required. These include multiple product line views, the feature model as the unifying view, an underlying product line meta-model that provides a schema for a product line repository, support for consistency checking among the multiple views, and support for feature-based product line derivation. This paper describes multiple-view modeling of software product lines, with particular emphasis on the feature modeling view, multiple-view UML meta-modeling for software product lines, variability management in the meta-model, and consistency checking between meta-model views. The paper then describes the requirements for tool support for product lines and product derivation, before describing a software prototype tool for this purpose and evaluating the effectiveness of the tool.

1 Introduction

Software variability management is a key challenge in developing software product lines and deriving products from the product line. In order to provide effective variability management and product derivation in software product lines, which is capable of being automated, the following is needed:

- a) Multiple product line views. A better understanding of the product line can be obtained by considering the different perspectives, such as requirements modeling, static modeling, and dynamic modeling, of the product line. A graphical modeling language such as UML helps in developing, understanding and communicating the different views.
- b) Feature model. One of the multiple views of the product line is the feature modeling view. The feature model is essential for both variability management and product derivation, because it describes the product line requirements in terms of commonality and variability, as well as defining the product line dependencies.

- c) Meta-model. A meta-model provides a unifying framework for the multiple views. Whereas the multiple views each need to use a different notation, a meta-model is represented in one notation. It contains the product line meta-classes and the relationships between the meta-classes, which allow consistency checking and assistance for product derivation.
- d) Product line repository. The meta-model is essential for tool support as it represents a schema for a product line repository, which stores the artifacts developed as a result of product line engineering.
- e) Consistency checking. Although a multiple view modeling approach helps in developing the product line, it is easy to introduce errors and inconsistencies in a multiple view model. It is therefore necessary to provide support for consistency checking among the multiple views.
- f) Product line derivation. The feature model is used to drive the process of product line derivation. By selecting a consistent set of features required for the individual product, the corresponding artifacts that realize those features are selected from the product line repository to constitute the product.

This paper starts by describing multiple-view modeling of software product lines. It then goes on to describe multiple-view meta-modeling for software product lines in UML, how variability is handled in the meta-model, and consistency checking between meta-model views. The paper then describes the requirements for tool support for product lines and product derivation, before describing a software prototype tool for this purpose and evaluating the effectiveness of the tool.

2 Multiple-View Models of Software Product Lines with UML

A multiple-view model for a software product line defines the different characteristics of a software family [8], including the commonality and variability among the members of the family [1, 5, 7, 11]. A multiple-view model is represented using the UML notation [4, 9]. The product line life cycle includes three phases for:

Product Line Requirements Modeling:

- Use Case Model View. The use case model view addresses the functional requirements of a software product line in terms of use cases and actors. Product line commonality is addressed by having kernel use cases, which are common and therefore directly reusable in all product line members. Product line variability is addressed by having optional and alternative use cases, which are used by some but not all product line members.

Product Line Analysis Modeling:

- Static Model View. The static model view addresses the static structural aspects of a software product line through classes and relationships between them. Kernel classes are common to all product line members, whereas optional and variant classes address product line variability.
- Collaboration Model View. The collaboration model view addresses the dynamic aspects of a software product line, which captures the sequence of messages passed between objects that realize kernel, optional, and alternative use cases.

- **Statechart Model View.** The statechart model view, along with the collaboration model view, addresses the dynamic aspects of a software product line. A statechart defines states and state transitions for each state dependent kernel, optional, and variant class.
- **Feature Model View.** A feature model view captures feature/feature dependencies, feature/class dependencies, feature/use case dependencies, and feature set dependencies. The feature model view is the key for managing variability in software product lines.

Product Line Design Modeling: During this phase, the software architecture of the product line is developed.

For software product lines, it is important to address how variability is modeled in each of the different views. A multiple-view model can be modified at specific locations referred to as variation points. More information on multiple-view modeling for software product lines is given in [6].

3 Multiple-View Meta-Model for Software Product Lines

Consistency checking between multiple views of a model is complex, one of the reasons being the different notations that are needed. An alternative approach [6, 10] is to consider consistency checking between multiple views at the meta-model level. The meta-model describes the modeling elements in a UML model and the relationships between them. The meta-model is described using the static modeling notation of UML and hence just uses one uniform notation instead of several. Furthermore, rules and constraints can be allocated to the relationships between modeling elements.

The multiple views are formalized in the semantic multiple-view meta-model, which depicts the meta-classes, attributes of each meta-class, and relationships among meta-classes. Relationships can be associations, compositions/aggregations (strong and weak forms of whole/part relationships), and generalization/specializations. A high level representation of the phases containing the views in this meta-model is shown in Figure 1. A phase is modeled as a composite meta-class that is composed of the views in that phase, as shown in Figure 1.

In the meta-class model, all concepts are modeled as UML classes. However, as the meta-classes have different semantic meaning, they are assigned stereotypes corresponding to the different roles they play in the meta-model. Thus in Figure 1, all the meta-classes represent the different views of a UML model and are assigned the stereotype «view». Meta-classes representing development phases are assigned the stereotypes «phase» as they represent the different phases of the OO lifecycle, Requirements Modeling, Analysis Modeling, and Design Modeling. Each view in Figure 1 can be modeled in more detail to depict the meta-classes in that view [10].

Fig. 1 depicts underlying relationships among multiple views in development phases of a software product line. The views in each phase are:

Requirements phase:

- Use case model: This model describes the functional requirements of a software product line in terms of actors and use cases.

Analysis phase:

- Class model: This model addresses the static structural aspects of a software product line through classes and their relationships.
- Statechart model: This model captures the dynamic aspects of a software product line by describing states and transitions.
- Collaboration model: This model addresses the dynamic aspects of a software product line by describing objects and their message communication.
- Feature model: This model captures the commonality and variability of a software product line by means of features and their dependencies.

The views of the design phase are described in [4]:

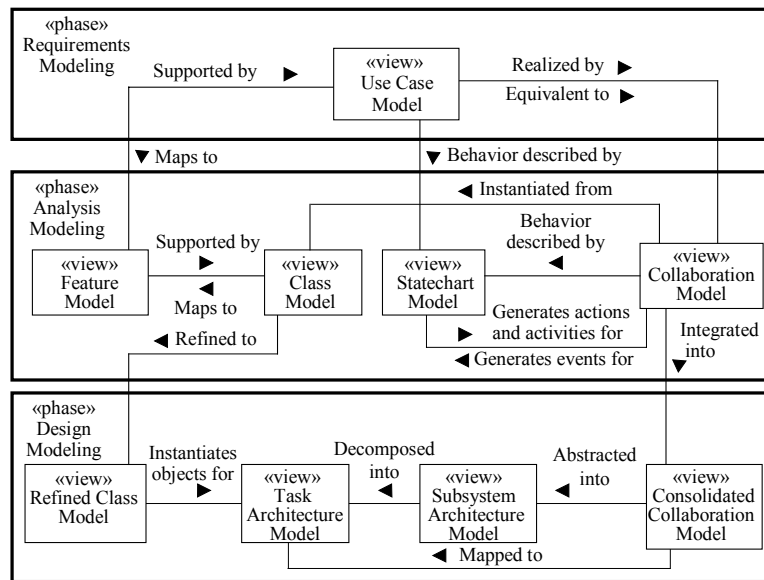


Fig. 1. High-level relationships between multiple views for a software product line

4 Consistency Checking between Multiple Views

Consistency checking rules are defined based on the relationships among meta-classes in the meta-model. The rules resolve inconsistencies between multiple views in the same phase or other phases, and to define allowable mapping between multiple views in different phases. To maintain consistency in the multiple-view model, rules defined at the meta-level must be observed at the multiple-view model level. Consistency checking is used to determine whether the multiple-view model follows the rules defined in the multiple-view meta-model.

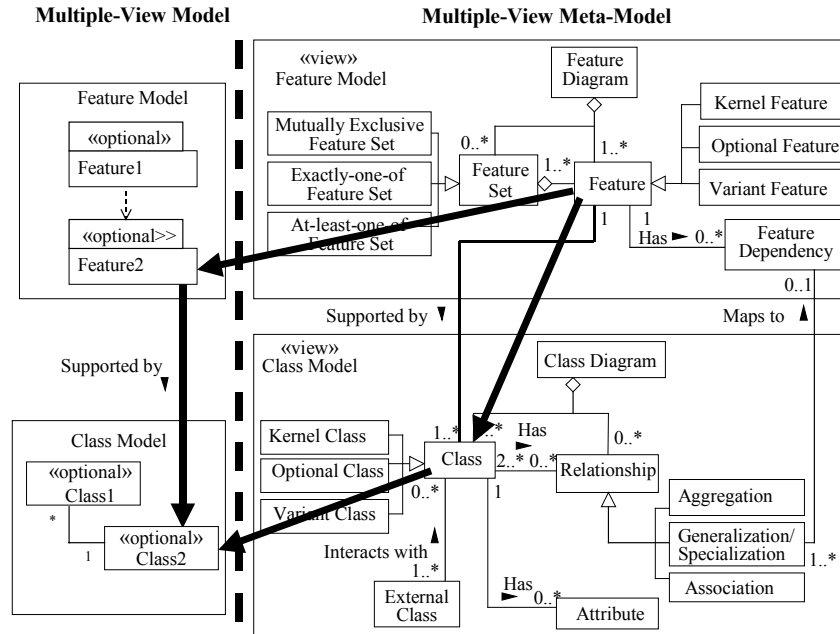


Fig. 2. Meta-model for feature and class model view

Fig. 2 depicts consistency checking between a feature in the feature model and a class in the class model. Suppose an optional class “Class2” supports an optional feature “Feature2.” Class2 and Feature2 in the multiple-view model are respectively instances of Class and Feature meta-classes in the multiple-view meta-model. There is a relationship between Class and Feature meta-classes, which is “each optional class in the class model supports only one optional feature in the feature model.” For the multiple-view model to remain consistent, this meta-level relationship must be maintained between instances of those meta-classes, that is, Class2 and Feature2. Consistency checking confirms that each optional class in the class model supports only one optional feature in the feature model.

5 Tool Support for Software Product Lines - Objectives

In order to support software variability management and product derivation in software product lines, the Product Line UML Based Software Engineering Environment (PLUSEE) has been developed. The objectives of the PLUSEE prototype are to:

- Provide tool support for representing the multiple graphical views of the product line modeling method.
- Provide a capability for consistency checking between the multiple views.
- Provide a capability for mapping the multiple views to a product line repository.

- d) Provide automated support for product derivation from the product line repository.
- e) Provide a product line independent environment. Thus the prototype should be capable of being used with multiple product line models.
- f) Because of limited resources and the need to focus those resources on the innovative parts of the project, use existing software tools where possible.

6 PLUSEE

The scope of the PLUSEE [10, 12] includes the product line engineering and product derivation phases (Fig. 3).

- a) Product line Engineering. A product line multiple-view model, which addresses the multiple views of a software product line, is modeled and checked for consistency between the multiple views. The product line multiple-view model and architecture is captured and stored in the product line reuse library.
- b) Product derivation. A target system multiple view model is configured from the product line multiple-view model. The user selects the desired features for the product line member (referred to as target system) and the tool configures the target system architecture.

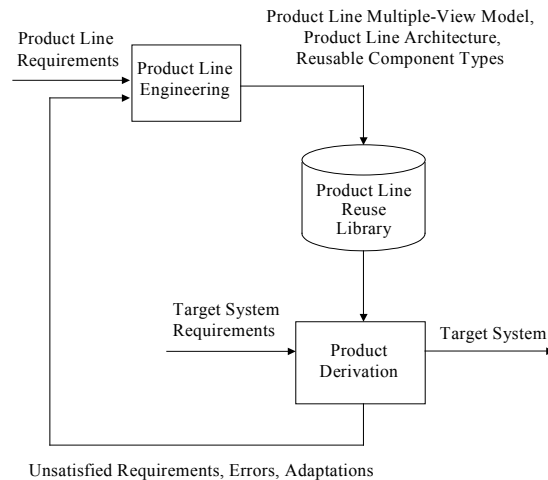


Fig. 3. Overview of PLUSEE

The PLUSEE represents second generation product line engineering tools which build on experience gained in previous research [2, 3]. PLUSEE builds on the experience gained with the earlier research with the Knowledge Based Software Engineering Environment (KBSEE). Whereas the KBSEE proof-of-concept prototype demonstrated that product line derivation from a product line feature model, architecture and components was feasible, it suffered from some serious limitations. Firstly, it used a Structured Analysis tool as a front end, and therefore had to rely on graphical editors

for data flow diagrams and entity-relationship diagrams, which lacked the richness needed to model object-oriented product lines. Secondly, although a product line repository was used, it was developed in an ad-hoc way and lacked the underlying meta-model to formally describe the product line artifacts and their relationships. This experience with KBSEE guided the following design decisions for the development of the PLUSEE:

- Both Rose and Rose RT Commercial CASE Tools were used as the graphical interface to this prototype. Rose supports all the views of the standard UML notation, but it does not generate an executable architecture from the product line multiple-view model. On the other hand, Rose RT generates an executable architecture from the product line multiple view model and simulates the product line architecture although it does not support all the views of the standard UML. To take advantages of Rose and Rose RT, two separate versions of PLUSEE, which are very similar to each other, were developed.
- The Knowledge Based Requirement Elicitation tool (KBRET) [2] and GUI developed in previous research were used without change.

6.1 Product Line Engineering

Fig. 4 depicts the overview of the product line engineering tools for PLUSEE, in which a product line engineer captures a product line multiple-view model consisting of use case, collaboration, class, statechart, and feature models through the Rose tools.

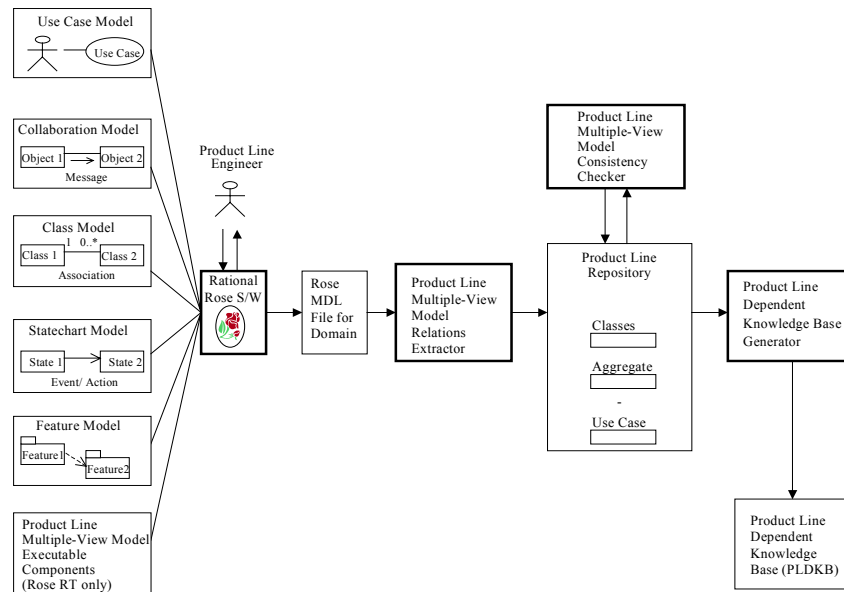


Fig. 4. Product line engineering tools for PLUSEE

- a) **Multiple-View Product Line Relations Extractor.** The multiple-view product line relations extractor generates product line relations from the multiple-view product line model. Rose and Rose RT save a multiple-view product line model in ASCII MDL and RTMDL files, respectively. In these files, information about the multiple-view model is stored with keywords. These keywords are used for extracting the information relevant to the multiple views of a software product line from the Rose MDL and Rose RTMDL files. The product line relations extracted are stored in an underlying tabular representation of the multiple views, which are later used for consistency checking and target system configuration. The product line relations are tool independent.
- b) **Product Line Model Consistency Checker.** The product line model consistency checker identifies inconsistencies between multiple views in the same phase or different phases. The rules for consistency checking between multiple views are checked against the product line relations extracted from the product line model. For example, the consistency checking rule in section 4, “each optional class in the class model must support only one optional feature”, is checked by the consistency checker using *Optional Class* relation ((a) of Fig. 5) and *Optional Feature Class Dependency* relation ((b) of Fig. 5), which are derived from the multiple-view model for the flexible manufacturing product line. The *Optional Class* relation contains optional classes derived from the product line static model. The *Optional Feature Class Dependency* relation defines a dependency between an optional feature and an optional class supporting the feature. To check the rule, the consistency checker confirms that each optional class in the *Optional Class* relation supports only one optional feature in the *Optional Feature Class Dependency* relation. For example, if the consistency checker finds an optional class that supports more than one optional feature, a kernel feature, or no feature at all, it generates a consistency error message for this rule.

Optional Class	
Part Scheduler	
Part Agent	
AGV Dispatcher	
Flexible Workstation Controller	

(a) Optional Class relation

Optional Feature	Optional Class
Flexible Manufacturing	Part Scheduler
Flexible Manufacturing	Part Agent
Flexible Manufacturing	AGV Dispatcher
Flexible Manufacturing	Flexible Workstation Controller

(b) Optional Feature Class Dependency relation

Fig. 5. Product line relations for consistency checker

- c) **Product Line Dependent Knowledge Base Generator.** The product line dependent knowledge base generator generates the product line dependent knowledge base from the product line relations. The product line dependent knowledge base contains information about classes, optional features, feature/feature depend-

ency, feature/class dependency, generalization/specialization relations among classes), aggregation relations among classes), and feature sets. The product line dependent knowledge base is used by KBRET to select target system features from the available optional features.

- d) **Knowledge Based Requirement Elicitation Tool.** The Knowledge Based Requirement Elicitation Tool (KBRET) is used to assist a user to select optional features of each target system. KBRET, which was developed in previous research [2], conducts a dialog with a human target system requirements engineer, presenting the user with the optional features available for selecting a target system. The user selects the features that will belong to the target system; KBRET reasons about feature/feature dependencies and then checks for feature set constraints such as mutually exclusive feature sets, exactly one-of feature sets, and one-or-more feature sets to resolve conflicts among features. Based on the selected features, KBRET determines the kernel, optional and variant classes to be included in this target system.

6.2 Product Derivation

In the product derivation phase of PLUSEE (Fig. 6), a Knowledge Based Requirement Elicitation tool (KBRET) is used to assist the human target system requirements engineer to select the optional features for the target system through KBRET GUI. Once the features are selected, KBRET reasons about the feature/feature dependencies to ensure that a consistent set of target system features are selected.

- a) **Target System Relations Extractor.** The target system relations extractor creates relations for a target system from the multiple-view product line relations. The goal is to tailor the product line multiple view model so as to configure a target system corresponding to the features selected for the target system. To extract target system relations, the extractor uses the optional and variant features that a user has selected through KBRET, as well as kernel features that are automatically selected for all product line members.
- b) **Target System MDL Generator.** The target system MDL generator was developed to create the Rose MDL file for a target system. Using the target system relations, the target system Rose MDL generator generates a Rose MDL file for a target system by changing the color of the modeling elements in the target system. A target MDL file for a target system is generated by changing the colors of target classes in the class model, target use cases in the use case diagram, target objects in the collaboration model, and target states in the statechart model. The changed color of target system multiple-view models (for example, yellow) is distinguished from the color of the original product line multiple-view model (for example, white).

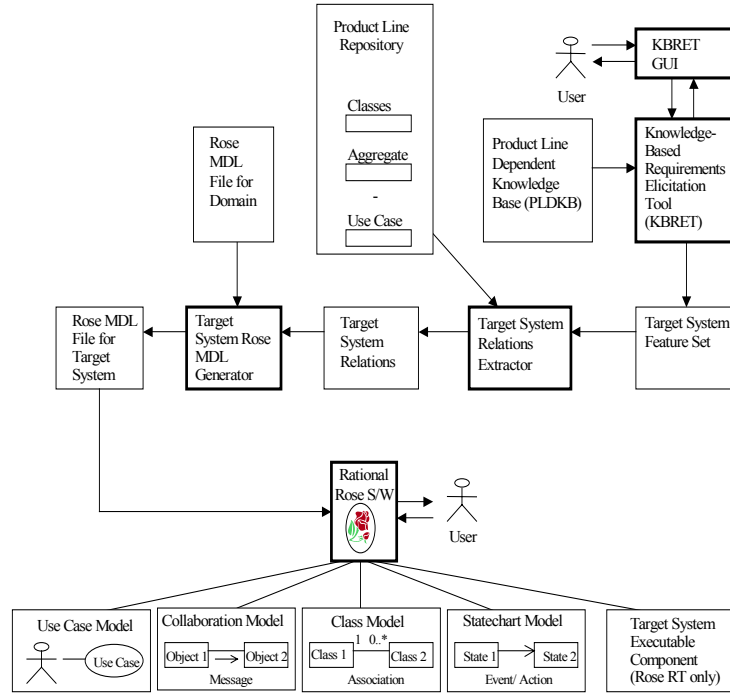


Fig. 6. Product derivation tools of PLUSEE

A Rose Real-Time executable model is a simulation of the target system, which is then executed and tested to determine whether the multiple-view model performs in accordance with the requirements.

7 Evaluation of PLUSEE

To evaluate this approach, the PLUSEE has been used in two case studies [10], a factory automation product line and an electronic commerce product line. The evaluation of the PLUSEE is conducted through the following validation procedure, which also identifies the activities performed by the human product line developer and the PLUSEE tool:

- Develop a multiple-view model of a software product line (Human).
- Map the multiple-view model to multiple-view model relations (Tool).
- Perform consistency checking of the multiple-view model relations (Tool).
- Implement multiple views using Rose Real-Time (Human).
- Configure target systems from the software product line (Tool and Human).

Each of the objectives listed above in section 6 was achieved as follows:

- a) Provide tool support for representing the multiple graphical views supported by the product line modeling method. This was achieved using the Rose graphical editors to support the multiple views. Rose was used to capture the multiple views; the underlying representation of each view was then extracted by our tools and mapped to the product line repository.
- b) Provide a capability for consistency checking between the multiple views. We developed a multiple view consistency checking tool for this purpose, which reported any inconsistencies among the views to the user.
- c) Provide a capability for mapping the multiple views to a product line repository. This was achieved by first using the open architecture provided by Rose to extract the information in the multiple views, mapping these views to an integrated set of data base relations that supported the multiple views, and then mapping these relations to a knowledge base repository. This was achieved using tools we developed for PLUSEE.
- d) Provide automated support for product derivation from the product line repository. This was achieved by developing the knowledge based requirements elicitation tool (KBRET) for this purpose. KBRET interacts with the product requirements engineer to derive the product from the product line repository.
- e) Provide a product line independent environment. Thus the prototype should be capable of being used with multiple product line models. Product line independence is achieved by treating all product line specific information as data and facts to be manipulated by the product line independent tools. To demonstrate product line independence, several different product lines have been modeled and products derived from them.
- f) Use existing software tools where possible. Both Rational Rose and Rational Rose RT were used for this research. It should be pointed out that the product line repository is CASE tool independent. To support a different UML modeling tool, it is necessary to develop a new version of the product line multiple view relations extractor. Thus, we developed two versions of the extractor, one for Rose and the other for Rose RT.

8 Conclusions

Software variability management is a key challenge in developing software product lines and deriving products from the product line. This paper has described how a UML-based multiple-view modeling approach for software product lines can be supported by a multiple-view UML meta-model for software product lines, which allows for variability management of the product line through the meta-model, and consistency checking between meta-model views. The paper then described the requirements for tool support for product lines and product derivation, before describing and evaluating a software prototype tool for this purpose. This research has demonstrated the viability of using UML-based methods and tools as a basis for variability management and product derivation in software product lines.

References

1. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, 2002.
2. H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I Tavakoli, "A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures," *J. Automated Software Engineering*, Vol. 3, Nos. 3/4, August 1996.
3. H. Gomaa and G.A. Farrukh, "Methods and Tools for the Automated Configuration of Distributed Applications from Reusable Software Architectures and Components", *IEE Proceedings – Software*, Vol. 146, No. 6, December 1999.
4. H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML," Addison-Wesley, 2000.
5. H. Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures, Addison-Wesley. To appear, July 2004.
6. Hassan Gomaa and Michael E. Shin, "Multiple-View Meta-Modeling of Software Product Lines" the Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Maryland, December, 2002.
7. K. C. Kang et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.
8. Parnas D., "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, March 1979.
9. J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual," Addison Wesley, Reading MA, 1999.
10. Michael E. Shin, "Evolution in Multiple-View Models in Software Product Families," Ph.D. dissertation, George Mason University, Fairfax, VA, 2002.
11. David M Weiss and Chi Tau Robert Lai, "Software Product-Line Engineering: A Family-Based Software Development Process," Addison Wesley, 1999.
12. Hassan Gomaa and Michael E. Shin, "A Multiple-View Modeling Approach for Variability Management in Software Product Lines" 8th International Conference on Software Reuse (ICSR 2004), Madrid, Spain, July, 2004.