Variability Modeling in Model-Driven Software Product Line Engineering

Hassan Gomaa[1], Michael E. Shin[2]

[1] Department of Computer Science, 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@ttu.edu

Abstract: This paper describes an approach for modeling variability in software product lines that are developed, and later evolved, using model-driven software product line development. This paper describes variability management in UML based multiple-view models of the software product line, which consist of developing use case and feature models during requirements modeling, static and dynamic models during analysis modeling, and component-based software architectures during design modeling. This paper also describes an underlying multiple-view meta-model for the software product line, with consistency checking between the multiple views.

1. Introduction

Although software product line (SPL) engineering [Clements02, Pohl05, Weiss99] is becoming increasingly used in industry, model-driven software product line engineering is less widely used. The most widely used model-driven software development approaches are used for single systems and are based around UML. Feature modeling, which is widely used in software product lines, is not used for single system modeling with UML. This paper describes an evolutionary model-driven software product line engineering approach using UML called PLUS (Product Line UML-based Software Engineering), in which feature modeling is fully integrated with the other modeling views. This paper also describes an underlying multiple-view meta-model for the SPL, with consistency checking between the multiple views.

The Object Management Group (OMG) advocates **model-driven architecture** in which "modeling is designing of software applications before coding". Software modeling approaches are now widely used in software development and have an important role to play in software product lines [Gomaa05]. Modern software modeling approaches, such as the Unified Modeling Language (UML) [Rumbaugh05], provide greater insights into understanding and managing commonality and variability by modeling product lines from different perspectives.

Kruchten [Kruchten95] introduced the 4+1 view model of software architecture, in which he advocated a multiple view modeling approach for software architectures, in

which the use case view is the unifying view (the 1 view of the 4+1 views). This paper describes a multiple view modeling approach for software product lines in which the unifying view is the feature model. In particular, feature modeling provides the added dimension of modeling variability in evolutionary software product lines.

After presenting an overview of the evolutionary SPL engineering approach in Section 2, this paper describes in Sections 3-5 how variability modeling is provided in each of the different modeling views to provide a better understanding of the SPL Multiple view meta-modeling, as the underlying representation of the multiple views, is described in Section 6 and consistency checking between the multiple views is described in Section 7. The approach is discussed in Section 8.

2. Evolutionary Software Product Line Engineering

The Evolutionary Software Product Line Engineering Process [Gomaa05] is a highly iterative software process that eliminates the traditional distinction between software development and maintenance. Furthermore, because new software systems are outgrowths of existing ones, the process takes a software product line (SPL) perspective; it consists of two main processes, as depicted in Fig. 1:

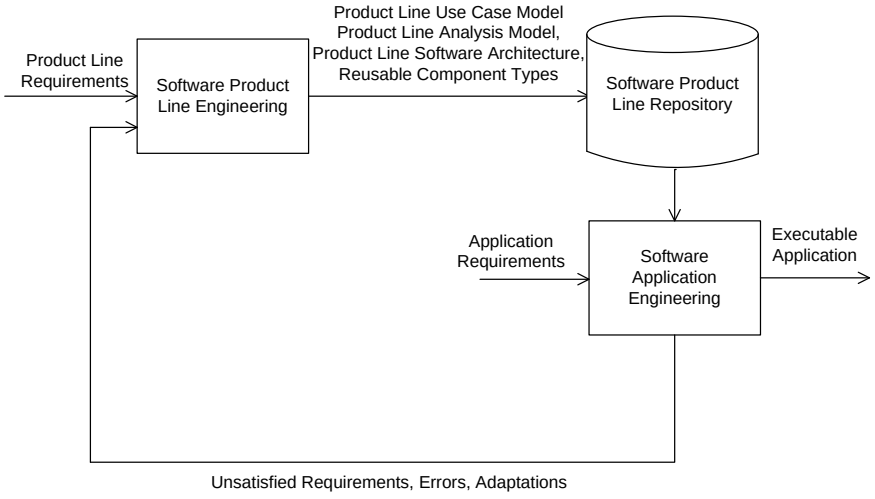

Figure 1: Evolutionary software product line process model

a) Product line Engineering (a.k.a. domain engineering). A product line multiple-view model, which addresses the multiple views of a software product line, is developed. The product line multiple-view model, product line architecture, and reusable components are developed and stored in the product line reuse library.

b) Software Application Engineering. A software application multiple-view model is an individual product line member derived from the SPL multiple-view model. The user selects the required features for the individual product line member. Given the features, the product line model and architecture are adapted and tailored to derive the application architecture [Gomaa06]. The architecture determines which of the reusable components are needed for configuring the executable application.

An important part of developing a SPL is commonality/variability analysis [Weiss99]. The kernel software architecture represents the commonality of the product line. Evolution is built into the software development approach because the variability in the software architecture is developed by considering the impact of each variable feature on the software architecture and evolving the architecture to address the feature. The development approach is a feature-based evolutionary approach, meaning that it addresses both the original development and subsequent post-deployment evolution. Being feature-based, the approach closely relates the evolution of the software architecture to the evolution of software requirements.

3. Variability in Requirements Modeling

3.1 Feature Variability Modeling

With multiple-view SPL modeling, it is possible to define the commonality and variability in each view. However, it is difficult to get a complete picture of the variability in the SPL because the variability is dispersed among the multiple views. To get a full understanding of the SPL variability, it is necessary to have one view that focuses entirely on variability and defines variability dependencies. That is the purpose of the feature modeling view.

Feature modeling [Kang90] is the unifying view for modeling variability in software product lines. Features are analyzed and categorized as common features (must be supported in all product line members), optional features (only required in some product line members), alternative features (a choice of feature is available) and prerequisite features (dependent upon other features). There may also be dependencies among features, such as mutually exclusive features. The emphasis in feature modeling is capturing the product line variability, which is characterized by optional and alternative features, since these features differentiate one member of the family from the others.

Features are used widely in product line engineering but are not used in UML. In order to effectively model product lines, it is necessary to incorporate feature modeling concepts into UML. Features can be incorporated into UML using the meta-modeling concept, in which features are modeled as meta-classes and given stereotypes to differentiate between <<common feature>>, <<optional feature>>, <<parameterized feature>> and <<alternative feature>>. Furthermore, feature groups, which place a constraint on how certain features can be selected for a product line member, such as mutually exclusive features, are also modeled using meta-classes and given stereotypes, e.g., <<zero-or-one-of feature group>> or <<exactly-

one-of feature group>> [Gomaa05]. Integrating feature modeling with UML is also described in [Vranic06].

An example of a feature model is given for a Banking SPL in Fig. 2, which depicts optional and parameterized features, as well as a feature group consisting of a default feature and alternative features. Optional features are the Greeting and Statement features. An example of an exactly-one-of feature group is the Language feature group, which consists of a default English feature as well as alternative Spanish, French or German features. The feature model could evolve by adding other Language alternatives or optional features such as a funds deposit feature.

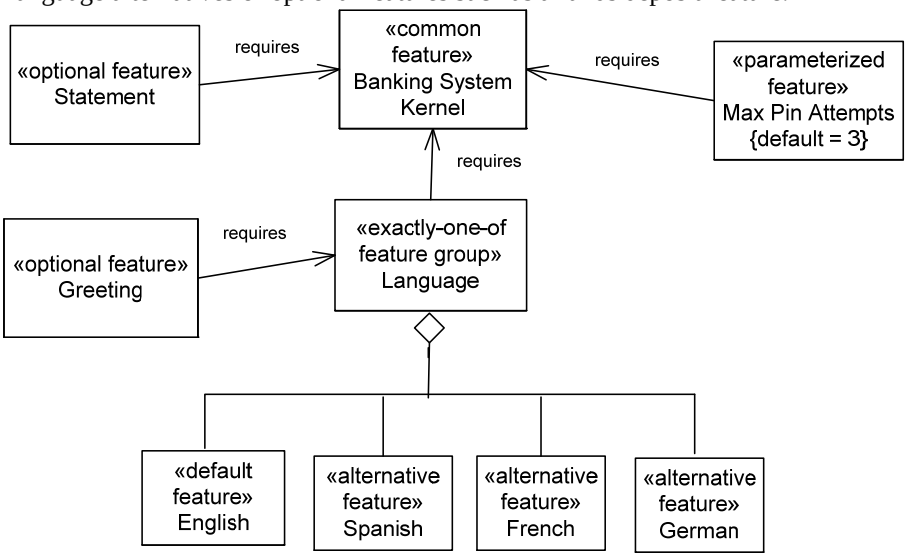

Figure 2 Example of Feature Model

3.2. Use Case Variability Modeling

The functional requirements of a system are defined in terms of use cases and actors [Rumbaugh05]. For a single system, all use cases are required. In a software product line, only some of the use cases, which are referred to as kernel use cases, are required by all members of the family. Other use cases are optional, in that they are required by some but not all members of the family. Some use cases may be alternative, that is different versions of the use case are required by different members of the family. In UML, the use cases are labeled with the stereotype <<kernel>>, <<optional>> or <<alternative>> [Gomaa05]. In addition, variability can be inserted into a use case through variation points, which specify locations in the use case where variability can be introduced [Jacobson97, Gomaa05, WebberGomaa04].

Use cases are used to determine the functional requirements of a system; they can also serve this purpose in SPL. Griss [Griss98] has pointed out that the goal of use case analysis is to get a good understanding of functional requirements whereas the goal of

feature analysis is to differentiate between commonality and variability. Use cases and features complement each other. Thus optional and alternative use cases are mapped to optional and alternative features respectively, while use cases variation points are also mapped to features [Gomaa05].

With use case modeling, it is more difficult to capture dependencies in use case variability, whereas a feature dependency can be explicitly captured in a feature model. For example, both Greeting and Language are use case variation points but Greeting depends on Language, which is explicitly captured in the feature model (Fig. 2) as a feature dependency. The same variability can be repeated in multiple use cases but is captured more effectively as one feature in a feature model. For example, the Language variation point is needed in several use cases but appears as one only feature in the feature model.

In the use case model for the Banking SPL (Fig. 3), the Statement feature (Fig. 2) is mapped to an optional Print Statement use case and the Language use case variation point, which is required in several use cases to model variability in the display language, is mapped to the Language feature group (Fig. 2). In an evolving SPL, a use case model could evolve by adding variation points to reflect further changes introduced through evolution and providing new optional or variant use cases.

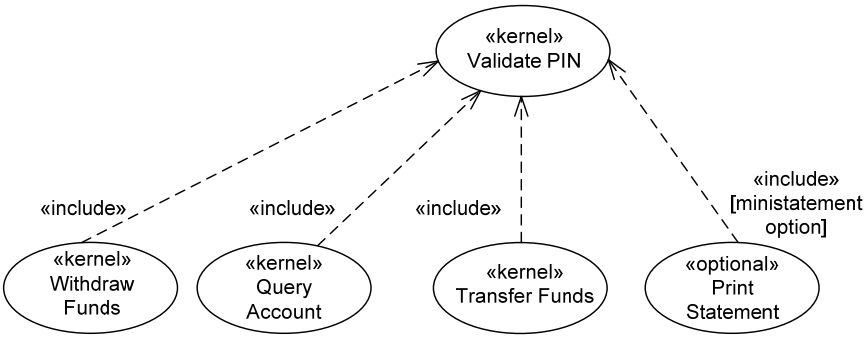

Figure 3 Product Line Use Cases

4. Variability in Analysis Modeling

4.1. Static Variability Modeling

In single systems, a class is categorized by the role it plays. Application classes are classified according to their role in the application using stereotypes, such as «entity class», «control class», or «interface class». In modeling software product lines, each class can be categorized according to its reuse characteristic using the stereotypes «kernel», «optional», and «variant». In UML 2.0, a modeling element can be described with more than one stereotype. Thus one stereotype can be used to represent the reuse characteristic while a different stereotype is used to represent the role played by the modeling element [Gomaa05]. The role a class plays in the application and the reuse characteristic are orthogonal.

For every feature in the product line, certain classes realize the functionality specified by the feature. For example, the Greeting feature is realized by the optional ATM Greeting entity class (Figure 4) and either the default English Display Prompts or a variant class, such as French Display Prompts.

Figure 4 Role and Reuse Stereotypes in Product Line Classes

4.2 Dynamic Interaction Variability Modeling

Variability modeling in dynamic interaction modeling is used to determine the dynamic impact of each feature on the software architecture of the software product line. This results in new components being added or existing components having to be adapted. The kernel system is a minimal member of the product line. In some product lines the kernel system consists of only the kernel objects. For other product lines, some default objects may be needed in addition to the kernel objects. The kernel system is developed by considering the kernel use cases, which are required by every member for the product line. For each kernel use case, an interaction diagram is developed depicting the objects needed to realize the use case.

Variability modeling during evolution of the product line starts with the kernel system and considers the impact of optional and alternative features [Gomaa05]. This results in the addition of optional or variant components to the product line architecture. For each optional or alternative feature, an interaction diagram is developed consisting of new optional or variant objects – the variant objects are kernel or optional objects that are impacted by the variable scenarios, and therefore need to be adapted. For example, the Greeting and Language features (Fig. 2) impact the interaction diagrams for the Validate PIN and Withdraw Cash use cases by needing the addition of the Greeting optional object and one of the variant Language objects.

4.3 State Machine Variability Modeling

When components are adapted for evolution, there are two main approaches to consider, specialization or parameterization. Specialization is effective when there are a relatively small number of changes to be made, so that the number of specialized classes is manageable. However, in SPL evolution, there can be a large degree of variability. Consider the issue of variability in control classes, which are modeling using state machines and depicted on statecharts [Harel96]. Modeling variability can be handled either by parameterized statecharts or specialized statecharts. The following discussion relates to evolution within a given state dependent component.

To capture product line variability and evolution, it is necessary to specify optional states, events and transitions, and actions. A further decision that needs to be made when using state machines to model variability is whether to use state machine inheritance or parameterization. The problem with using inheritance is that a different state machine is needed to model each alternative or optional feature, or feature combination, which rapidly leads to a combinatorial explosion of inherited state machines. For example, with only three features that could impact the statechart, there would be eight possible feature and feature combinations, resulting in eight variant statecharts. With 10 features, there would be over 1000 variant statecharts. However, 10 features can be easily modeled on a parameterized statechart as 10 feature dependent transitions, states, or transitions.

It is often more effective to design a parameterized state machine, in which there are feature-dependent states, events, and transitions. Optional transitions are specified by having an event qualified by a feature condition, which guards entry into the state. Thus, to realize the Minute Plus feature (which adds a minute to cooking time and starts cooking if oven is off), a Minute Pressed feature dependent state transition is introduced, which is guarded by the feature condition [minuteplus], as depicted in Figure 5. Similarly, there can be feature-dependent actions, such as Switch On and Switch Off in Figure 5, which are only enabled if the Light feature condition is True. Thus the feature condition is True if the optional feature is selected for a given product line member, and false if the feature is not selected. The impact of feature interactions can be modeled very precisely using state machines through the introduction of alternative states or transitions.

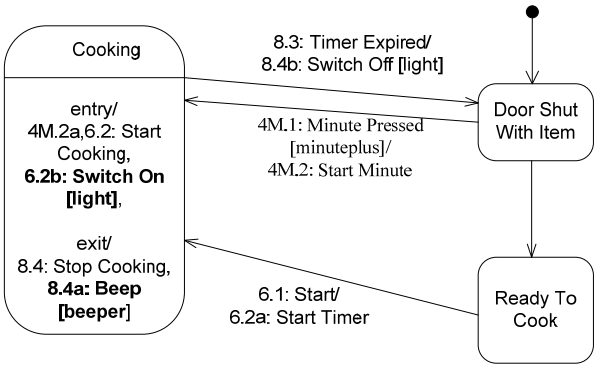

Figure 5 Feature Dependent Transitions and Actions

5. Modeling Variability in Software Architectures

A software architecture [Bass03] separates the overall structure of the system, in terms of components and their interconnections, from the internal details of the individual components. Software components can be effectively modeled in UML 2.0 with structured classes and depicted on composite structure diagrams [Rumbaugh05].

Structured classes have ports with provided and required interfaces. Structured classes can be interconnected via connectors that join the ports of communicating classes.

For a SPL component-based software architecture for, it is necessary to specify the interface(s) provided by each component and the interface(s) required by each component. This capability for modeling component-based software architectures is particularly valuable in product line engineering, to allow the development of kernel, optional and variant components, “plug-compatible” components, and component interface inheritance. It is highly desirable, where possible, to design components that are plug-compatible, so that the required port of one component is compatible with the provided ports of other components to which it needs to connect [Gomaa05].

Consider the case in which a producer component needs to be able to connect to different alternative consumer components in different product line members, as shown in Figure 6. The most desirable approach, if possible, is to design all the consumer components with the same provided interface, so that the producer can be connected to any consumer without changing its required interface. For example, the Customer Interaction component could be connected to either the English Display Prompts component or the French Display Prompts component (which correspond respectively to the default English feature and alternative French feature), given the two versions of prompt component provide the same interface. As the SPL evolves new producers can communicate with the consumer, or new alternative consumer components could be added, given they are compatible with the provided interface. When plug-compatible components are not practical, an alternative component design approach is component interface inheritance [Gomaa05].

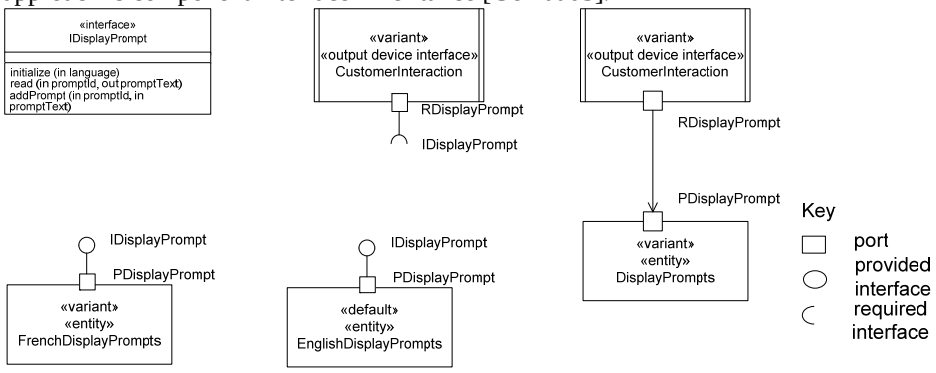

Figure 6 Design of Plug-compatible Components

6. Multiple View Meta-Model for Software Product Lines

The relationships between the multiple views of a model are complex, one of the reasons being the different notations that are needed. An alternative approach [GomaaShin08] is to consider the relationships between the multiple views at the meta-model level. The meta-model describes the modeling elements in a UML model

and uses one uniform notation instead of several. Furthermore, rules and constraints can be allocated to the relationships between modeling elements. The meta-model can then be used for consistency checking among the multiple views [GomaaShin08]. Consistency checking in UML diagrams is also described in [Chen09].

The multiple views are formalized in the meta-model, which depicts the meta-classes, attributes of each meta-class, and relationships among meta-classes. Relationships can be associations, compositions or aggregations (strong and weak forms of whole/part relationships), and generalization/specializations. As the meta-classes have different semantic meaning, they are assigned stereotypes corresponding to the different roles they play in the meta-model. Thus, meta-classes that represent different views of a UML model are assigned the stereotype «view». Meta-classes representing different phases of the OO lifecycle are assigned the stereotypes «phase» e.g., Requirements Modeling, Analysis Modeling, and Design Modeling. A phase is modeled as a meta-class that is composed of the views that are developed as part of that phase.

Figure 7 depicts a multiple view meta-model for SPL. The meta-model depicts the six views as meta-classes, each with stereotype «view». The Feature Model and Use Case Model are views that are part of the Requirements Modeling phase. The Static Model, State Machine Model, and Dynamic Interaction Model, are part of the part of the Analysis Modeling phase. The Software Architecture Model is developed in the Design Modeling phase. Each view in Figure 7 can be modeled in more detail to depict the meta-classes in that view, as described in [GomaaShin08].

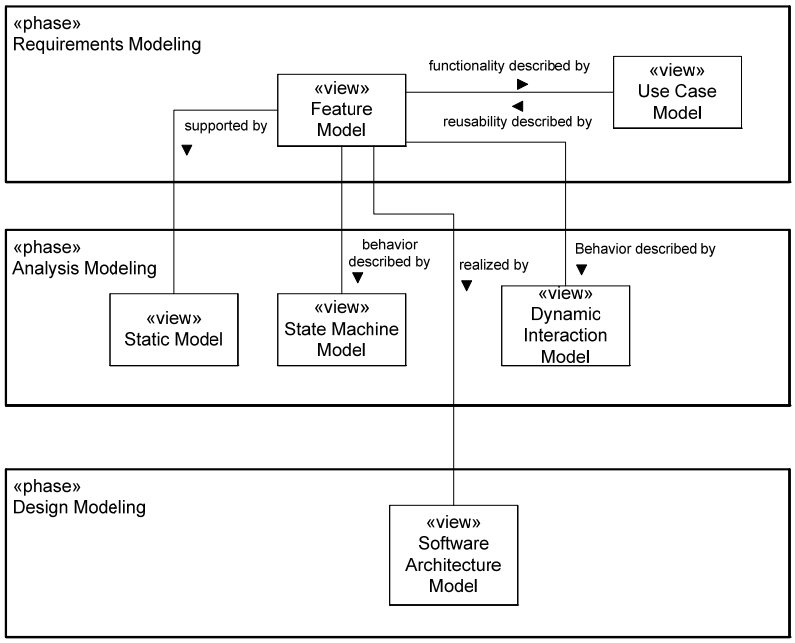

Figure 7: Multiple View Meta-model for Software Product Lines

The multiple view meta-model for SPL depicted in Figure 7 is feature model centric, in that it shows the feature model as the unifying view and its relationship with the other five views. There are of course relationships among these five views, as described in [GomaaShin08]. However, this paper concentrates on the relationships with the feature model.

7. 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.

Figure 8 depicts a meta-model describing the relationships between the feature and class model views in Figure 7. Figure 8 also 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. The relationship between Class and Feature meta-classes 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 Class2 and Feature2 of the meta-classes. Consistency checking confirms that each optional class in the class model supports only one optional feature in the feature model.

8. Discussion

This paper has described how feature modeling is the unifying view in a multiple view SPL. In particular, a feature model is a more effective approach for modeling the variability in a SPL in the form of features, feature groups and feature dependencies. Thus the feature model in figure 2 gives a much clearer depiction of the variability in the SPL than the use case model. However, the use case model is much better at describing the overall functionality of the SPL. By explicitly relating the feature model to the use case model in feature/use case tables [Gomaa05], both the variability and the functionality of the SPL can be captured and related to each other. Similarly, feature/class tables explicitly identify how the variability in requirements is mapped to variability in the software architecture in the form of optional and variant classes, and feature based class parameters. Variability in state machines is captured by feature dependent states, transitions and actions.

In short, by using features and feature modeling, variability in each modeling view can be explicitly depicted and associated with the feature model. Thus even though features have no semantics [Czarnecki05], they have an important role to play in modeling variability and relating this variability to each of the views in multiple view

models. In comparison to other feature modeling approaches [e.g., Kang90, Griss98, Czarnecki05, Vranic06], the approach described in this paper explicitly relates feature modeling (at both the modeling and meta-modeling levels) to other modeling views that span the requirements, analysis, and design phases of the SPL life cycle.

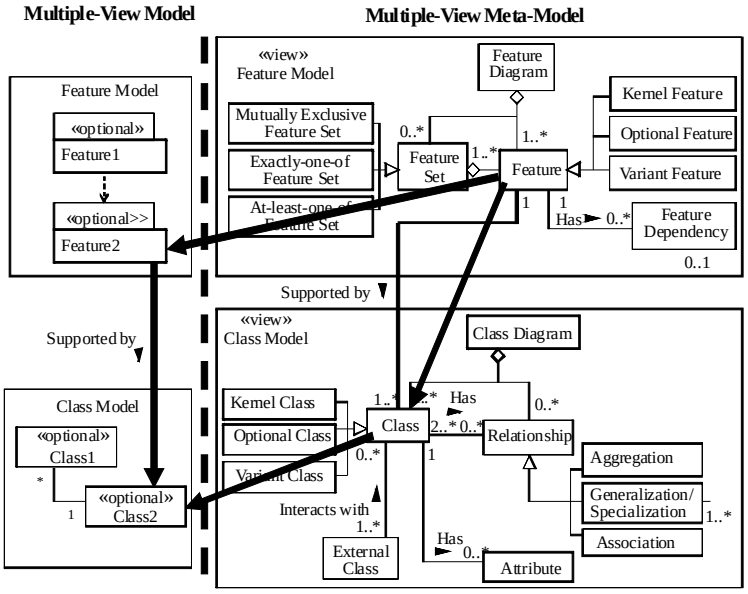

Figure 8 Consistency checking in meta-model for feature and class model view

9. Conclusions

This paper has described an approach for modeling variability in software product lines, which are developed and later evolved using model-driven SPL development. This paper has described variability modeling in UML based multiple-view models of the SPL. These models consist of use case and feature models during requirements modeling, static and dynamic models during analysis modeling, and component-based software architecture models during design modeling. Feature modeling is the unifying view for model driven development and evolution of SPLs, since it provides an additional dimension for modeling variability.

This paper has also described an underlying meta-model for software product lines, which allows consistency checking between the multiple views. Tool support for this approach is described in [GomaaShin08] in which the multiple-view meta-model is mapped to relational tables, which are then checked for consistency. Related research describes separation of concerns for SPL [ShinGomaa10], feature-based testing of SPL [OlimpiewGomaa09], adaptive SPL [GomaaHussein07], and feature modeling for service-oriented architectures [AbuMatarGomaa10].

References

[AbuMatarGomaa10] M. Abu-Matar, H. Gomaa, M. Kim, A. Elkhodary, Feature Modeling for Service Variability Management in Service-Oriented Architectures, Proc. 22nd Conf. on Software Engineering and Knowledge Engineering (SEKE), Redwood, Calif., July 2010.

[Bass03] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison Wesley, Reading MA, Second edition, 2003.

[Chen09] Z. Chen and G. Motet, "A Language-Theoretic View on Guidelines and Consistency Rules of UML," Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, Enschede, Netherlands, Springer 2009.

[Clements02] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison Wesley, 2002.

[Czarnecki05] K. Czarnecki and M. Antkiewicz, Mapping Features to Models: A Template Approach Based on Superimposed Variants". 4th International Generative Programming Conference, 2005, Springer LNCS 3676, pages 422–437, Tallinn, Estonia, October 2005.

[Gomaa05] Gomaa, H., "Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures", Addison-Wesley, 2005.

[Gomaa06] H. Gomaa & M. Saleh, "Feature Driven Dynamic Customization of Software Product Lines", Proc. Intl. Conf. Software Reuse, Turin, Springer LNCS 4039, June 2006.

[GomaaHussein07] H. Gomaa and M. Hussein, "Model-Based Software Design and Adaptation", Proc. ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Minneapolis, MN, May 2007.

[GomaaShin08] H. Gomaa and M.E. Shin, "Multiple-View Modeling and Meta-Modeling of Software Product Lines", Journal of IET Software, Vol. 2, Issue 2, pp. 94-122, April 2008.

[Griss98] Griss, M., J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB." In *Fifth Intl. Conf. on Software Reuse: Proc: June 1998, Victoria, BC, Canada*, P. Devanbu and J. Poulin (eds.), pp. 1–10. Los Alamitos, CA: IEEE Computer Soc. Press.

[Harel96] Harel, D. and E. Gary, "Executable Object Modeling with Statecharts", Proc. 18th International Conference on Software Engineering, Berlin, March 1996.

[Jacobson 97] Jacobson, I., M. Griss, and P. Jonsson. 1997. *Software Reuse: Architecture, Process and Organization for Business Success*. Reading, MA: Addison-Wesley.

[Kang 90] Kang K. C. et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[Kruchten95] Kruchten P., "Architectural Blueprints - The 4+1 View Model of Software Architecture", IEEE Software 12 (6), November 1995.

[Olimpiew Gomaa09] E. Olimpiew and H. Gomaa, "Reusable Model-Based Testing", Proc. 11th Intl. Conf. on Software Reuse, Falls Church, VA, Springer LNCS 5791, Sept. 2009.

[Pohl05] K. Pohl et al, "Software Product Line Engineering: Foundations, Principles and Techniques", Springer, 2005.

[Rumbaugh05] J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual," Second Edition, Addison Wesley, Reading MA, 2005.

[ShinGomaa10] M.E. Shin and H. Gomaa, "Separating Application and Security Concerns in Modeling Software Product Lines", in Applied Software Product Line Engineering, edited by K.C. Chang, V. Sugumaran, and S. Park, CRC Press, 2010.

[Vranic06] V. Vranic and J. Snirc, "Integrating Feature Modeling into UML", Proc NODe 2006, Erfurt, Germany, September, 2006.

[WebberGomaa04] D. Webber and H. Gomaa, "Modeling Variability in Software Product Lines with the Variation Point Model", Journal of Science of Computer Programming, Volume 53, Issue 3, Pages 305-331, Elsevier, December 2004.

[Weiss99] D M Weiss and C T R Lai, "Software Product-Line Engineering: A Family-Based Software Development Process," Addison Wesley, 1999.