in Proceedings SERPS'04, Linköping, Sweden, Oct 21-22, 2004.

Marrying Features and Use Cases for Product Line Requirements Modeling of Embedded Systems

Magnus Eriksson
Alvis Hägglunds AB
SE-891 82 Örnsköldsvik, Sweden
MagnusEriksson@alvishagglunds.se

Jürgen Börstler
Umeå University
Dept. of Comp. Science
SE-901 87 Umeå, Sweden
jubo@cs.umu.se

Kjell Borg
Alvis Hägglunds AB
SE-891 82 Örnsköldsvik Sweden
KjellBorg@alvishagglunds.se

Abstract

Software intense defense systems, for example vehicles, are developed in short series and are expected to have an extremely long life span. Furthermore, these systems are always customized for different customer needs. For an organization to be competitive in a market like this it is important to achieve high levels of reuse between different customer projects and to be efficient in maintaining the different systems once developed. An interesting approach to address issues like these is known as product line development. Unfortunately, most published approaches for product line development today has software, not system, focus. In this paper we propose an approach to overcome this issue by introducing domain modeling with features and use cases as part of the system requirements process.

1. Introduction

Software intense defense systems, for example vehicles, are developed in short series and are expected to have an extremely long life span, often 30 years. Furthermore, these systems are always customized for different customer needs. For an organization to be competitive in a market like this it is important to achieve high levels of reuse between different customer projects and to be efficient in maintaining the different systems once developed. An interesting approach to address issues like these, which has gained considerable attention both by industry and academia over the last few years, is known as software product line development.

The basic idea of the software product line approach, or product family approach as it is sometimes referred to, is to use business domain knowledge to identify common parts within a family of products and to separate them from the differences between the products. The commonalties are then used to create a product platform that can be used as a common baseline for all products within the product family. Studies have shown that organizations can yield considerable improvements in productivity, time to market, product quality and customer satisfaction by applying this approach [Cle01].

Unfortunately, most published approaches for product line engineering only address the software problem and not the systems problem, which is the main focus in our domain. The systems problem differs from the pure software problem, as it would typically involve a broader set of requirements [RUP03]. This broader set of requirements must be addressed, allocated to and further detailed for the different subsystems that the system is composed of before subsequent detailed design can proceed [Dor97].

Our approach to tackle the lack of systems engineering support for embedded software product line development is to introduce domain modeling and requirements reuse within the systems engineering process. By introducing such product family concepts on the system level, we argue that organizations have better chances of succeeding when trying to introduce product line concepts for their embedded software development.

In this paper, we discuss a promising approach to system level domain modeling that utilizes feature models [Kan90] and use cases [Jac97] which are integrated into a coherent two-layer product family model. The main contributions of this paper are the clarification of the relationship between features and use cases and an improved approach to describe variant behavior in use case descriptions.

The paper is structured as follows: Section 2 provides a short introduction to feature modeling.

Section 3 provides a short introduction to use case modeling and describes the concept of system requirements flowdown. Section 4 discusses approaches to manage variability in use case models. Section 5 describes our approach to domain modeling with features and use cases. Section 6 discusses experiences applying the approach. Section 7 summarizes the paper, draw conclusion and mentions some future work in the area.

2. Feature Modeling

Kang first suggested feature models in 1990 as part of the Feature Oriented Domain Analysis (FODA) [Kan90] and they were later refined in the Feature Oriented Reuse Method (FORM) described in [Kan98].

A feature is defined as a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems in FODA. Features provide an abstract view of the product family that is easily understood by stakeholders [Cha01]. In feature models, features are organized into trees of AND and OR nodes that represent the commonalties and variabilities of the modeled domain. General features are located at the top of the tree and more refined features are located below. The root of the tree represents the complete system. Originally, FODA described "mandatory" and "optional" features that may have the relations "requires" and "excludes" to other features. All mandatory features are available in all systems built within the family. Optional features represent variability within the family that may or may not be included in systems built within the family. The relations "requires" and "excludes" are used to support consistency checking of the model when features are selected to be included in a system. A "requires" relationship indicates that a feature depend on some other feature to make sense in a system. An "excludes" relationship between two features indicate that the features can not both be included in the same system. Feature models can provide a very compact notation for describing complex relationships between different features within a domain.

Figure 1 shows a simple feature model describing a personal identification system. All systems in the example domain require use of a chip card. PIN codes or biometric identification are optional features in this domain and can be selected independently of each other. If the biometric identification feature is selected in a system, a mandatory selection between the alternative features fingerprint recognition and voice recognition must be done. Alternative features have a relationship equivalent to the "excludes" relationship mentioned above, however using a more compact notation.

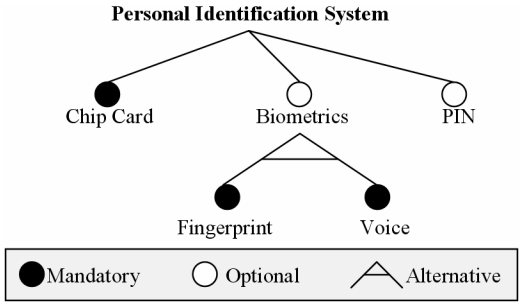

Figure 1: An example of a simple FODA feature model

Important improvements and clarifications of the FODA feature modeling approach have been proposed at the Nokia Research Center by Fey et al. [Fey02]. Fey et al. proposed the feature meta-model shown in Figure 2 to come to terms with several shortcomings of the original method. One example of such a shortcoming is that original FODA has no defined mechanism to specify the relation "at-least-one-out-of-many". To overcome this problem, Fey et al. defined the concepts of "Pseudo features" and the "provided-by" relation to capture the missing mechanism.

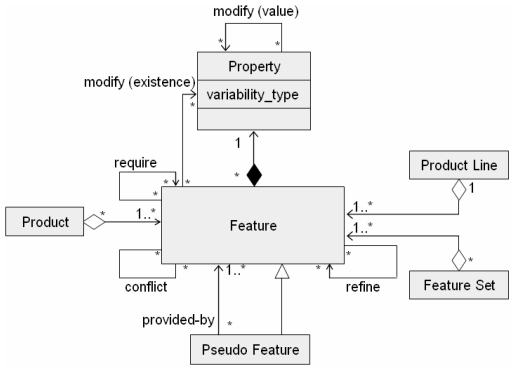

Figure 2: The Nokia Feature Meta-model from [Fey02]

3. Use Case Modeling

The concept of use cases was first introduced by Ivar Jacobson in the early 1990s and has today become a widely accepted and used requirement modeling technique, both in the software and the systems engineering communities. Use case modeling is a functional decomposition technique that provides a semi formal framework for structuring requirements.

The Unified Modeling Language (UML) version 1.5 standard [UML03] specifies that a use case is used to define the behavior of a system or some other semantic entity without revealing the entity's internal structure. A good use case should address one complete and well-defined goal that the user wants to accomplish by interacting with the system [Coc03].

Many software development process frameworks, for example the Rational Unified Process [Kru00], are based on use cases. Use cases can be the base on which project managers make time estimates and release plans. Designers can relate specific designs to requirements through the behavior defined in use cases. User interface designers can design and link their designs to requirements through use cases. Testers can create test scenarios based on the behavior and conditions described by use cases.

3.1 The UML Use Case Diagram

Use case diagrams show the relationships between actors and use cases within a system. Use case diagrams are typically used to provide an overview of the system functionality and to visualize system boundaries and interfaces to external system.

Actors represent a coherent set of roles that users of the modeled system can play. Actors can represent both human users and other external systems that the modeled system interacts with. Actors have two allowed relations to other use case model elements according to the UML version 1.5 standard [UML03]:

1. Actors can have a "generalization" relationship to another actor. This means that a child actor can communicate with at least the same use cases as its parent.
2. Actors can have an "association" relationship with one or more use cases. This means that instances of a use case and instances of an actor communicate with each other.

Beside the association relation to actors mentioned above, use cases are allowed to have the following relations to other use cases according to the UML standard:

1. Use cases can have a "generalization" relationship to other use cases. This means that a child use case inherits all associations and features from the parent and may also add new associations and features.
2. Use cases can have an "include" relationship to other use cases. This means that the behavior of the included use case is inserted in the base use case. The base use case may only depend on the result of the included use case, not its structure.
3. Use Cases can have an "extend" relationship to other use cases. This means that the behavior of a base use case is augmented by the behavior of another use case if a condition is fulfilled. The relationship also includes a number of extension points within the base use case that define where additional behavior fragments should be inserted.

3.2 Use Case Descriptions

The contents of a use case specify a set of scenarios describing sequences of interaction between the system and its actors. There is, however, no standardized way of describing these use case scenarios. A number of techniques, such as sequence diagrams, activity diagrams and state machines, can be utilized [UML03]. However, the most common way to describe use case behavior is to use natural language. A use case specification should define all relevant scenarios [Coc03]. All possible alternatives and failure scenarios should be captured on a level of abstraction that is appropriate for the current level of system description.

3.3 System Requirements Engineering with the RUP SE Use Case Flowdown Activity

While system level requirements are being developed, systems engineers start to consider what elements should make up the system architecture. When the first version of the system requirements specification is finished, there is usually also a preliminary first level system partitioning available [Dor97].

The next step of the systems engineering process is usually referred to as allocation. Allocation is the process of linking each system level requirement to one or more element in the system architecture; this is done by determining, for each system requirement, which elements of the system architecture will participate in meeting the requirement.

The next step of the process is sometimes referred to as requirements flowdown. The flowdown activity consists of writing derived requirements specifications for each element of the systems architecture based on the allocated system requirements. Each system requirement must have at least one subsystem requirement derived from it for each subsystem it was allocated to [Dor97].

The IBM-Rational approach to systems engineering is called RUP SE [RUP03], and it is a derivate of the Rational Unified Process (RUP) [Kru00] that better addresses the systems engineering problem. RUP SE, as traditional RUP, is a use case driven approach. The

RUP SE use case flowdown activity is a way to derive functional requirements to the elements of the system architecture [RUP03]. The activity can be summarized to consist of the following steps:

1. Identify architecturally significant use cases.
2. For each architecturally significant use case, develop its "Black box flow of events." The "Black box flow of events" is a tabular notation for describing use case scenarios as shown in Figure 3. Each step in the scenario has a step identifier, a description of some actor action, a description of some system response to that actor action and possibly some associated non-functional requirements to the system response. Descriptions of system actions should be black box, that is, not reference architectural elements [RUP03].

Step	Actor Action	Black Box	Black Box Budget Requirement
1	This use case begins when the **Clerk** pushes the New Sale button.	The **System** brings up new sale clerk and customers screens and enables the scanner.	Total response time is 0.5 second.
2	The **Clerk** scans the items and enters the quantity on the keyboard.	For each scanned item, the **System** displays the name and price.	Total response time is 0.5 second.
3	The **Clerk** pushes the Total button.	The **System** computes and displays on the screen the total of the item prices and the sales taxes.	Total response time is 0.5 second.
4	The **Clerk** swipes the credit card.	This use case ends when the **System** validates the credit card, and: If the credit card is valid, the **System** prints out a receipt, updates the inventory, sends the transaction to accounting and clears the terminal. If the credit card is not valid, the **System** returns a reject message	Total response time is 0.5 second.
		If the credit card is not valid, the **System** returns a reject message	Total response time is 30 seconds.

Figure 3: An example RUP SE Black box description from [RUP03]

3. Perform object-oriented analysis and design to identify the system architecture.
4. Revisit the Black box flow of events and decompose each Black box step into a number of White box steps, by documenting how the architectural elements collaborate to solve each black box step. In the example shown in Figure 4, the first Black box step of Figure 3 is decomposed into three White box steps describing how the system entities "***Point-of-Sales Interface***" and "***Order Processing***" collaborate to solve the step.
5. Sort the White box steps by subsystem, process and hardware unit to create use case surveys for each system element.

Step	Actor Action	Black Box	Black Box Budget Requirement	Subsystem White Box	White Box Budget Requirements
1	This use case begins when the **Clerk** pushes the New Sale button.	The **System** brings up new sale clerk and customers screens and enables the scanner.	Total response time is 0.5 second.	**The Point-of-Sale Interface** clears the transaction, brings up new sales screens, and requests that **Order Processing** start a sales list.	1/6 second.
				Order Processing starts a sales list.	1/6 second.
				Point-of-Sale Interface enables the scanner.	1/6 second.

Figure 4: An example RUP SE White box step description from [RUP03]

4 Managing Variability in Use Case Models

The UML Use Case meta-model provides poor assistance in modeling variability [Maβ02]. This is unfortunate, since variability management is a key success factor in software product line development. If it is not possible manage information about variants within the product family, it is not going to be possible to design a product line architecture that will be able to capture the whole family either.

4.1 Managing Variability in Use Case Diagrams

A number of suggestions how to address this issue are described in the literature. Von der Maβen and Lichter suggest that the UML use case meta-model should be extended by two new relationships, "Option" and "Alternative", to be able to model variability in use case diagrams [Maβ02]. John and Muthig suggest that the UML "Stereotype" mechanism (see [UML03]) could be used to model variability in use case diagrams [Joh02]. Jacobson et al. suggest use of the "generalization" and "extend" relationships to model variability in UML use case diagrams [Jac97]. They suggest that the generalization relation typically should

be used when there is a sequence of common behavior that can be defined and reused by several other use cases. According to Jacobson et al. the "extends" relationship could be used in the following two different ways:

1. The "extends" relationship can be used to add behavior to a use case that is already "complete" in it self.
2. The "extends" relationship can be used to insert variant behavior at a specified extension point that will be executed when the use case is executed. Using this mechanism makes the use case behave as a template that must be filled in before use.

We propose that UML diagrams be used according to Jacobson et al. recommendations. However, not to model variability within the product family, but to describe relationships between individual use cases. The problem we see with using UML use case diagrams for describing variants within the system family is that they soon get cluttered to a degree where it is impossible to see the high level view of the family.

4.2 Managing Variability in Use Case Descriptions

It is however not enough to only manage variability among whole use cases, it must also be possible to specify variant behavior within use cases descriptions.

Definition of these variations can, for example, be done by specifying parameters within the use case descriptions as was done by Jacobson et al. in [Jac97]. There have also been some proposals in the literature on how to describe variant behavior in use case scenarios, one example are extended templates with XML-like tags around variant steps. Trigaux and Heymans have provided an overview of some of these approaches in [Tri03].

The problems we see with these existing approaches are twofold. First of all, they do not recognize the difference between global variation points that can be instantiated for the whole domain model, and local variation points that can only be instantiated for the current element in the domain model. This could lead to inconsistency problems when deriving product specifications from the domain model. Secondly, approaches to specify variant behavior in use case scenarios do not provide any mechanism to enable traceability of those variants to the system architecture.

4.3 Change Cases

Change cases, which were first suggested by Ecklund et al. in 1996 [Eck96], are a way to identify and articulate anticipated changes to a system over its foreseeable lifetime. Ecklund et al. defines a change case to consist of two parts, a use case with new or revised scenarios and a set of existing use cases. In other words, change cases are basically use cases that model possible future functional requirements of a system. However in change cases, the use case meta-model is extended to allow the relation "Impact Link". Impact links provide traceability to use cases that would be affected if the change case where to be implemented in the system.

Figure 5 shows a partial use case model for a banking service. The use case model includes a change case that models the possible future service "InternetTransfer". According to the example, implementing such a service would impact how customers are identified and how information is displayed to customers. Figure 5 also show use of the UML use case relations "include" and "extends" as mentioned in Section 4.1.

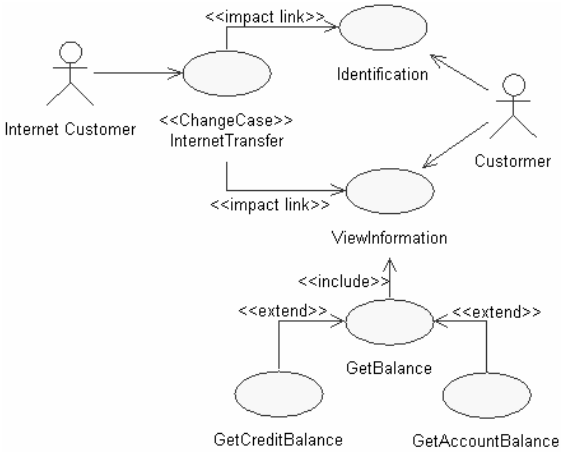

Figure 5: An example change case in a partial banking service use case model in UML

Modeling change cases in software product lines are useful for mainly two reasons. First of all, change cases assist architects and designers in planning a robust architecture that can effectively capture future functional requirements on the system. Secondly, the traceability links of the change cases provide useful support for change impact analysis when considering implementation of potential changes to the system.

4.4 A Two-Layer Product Line Use Case Model

According to Clements and Northrop, product line wide requirements are an important core asset that should be maintained separately from product specific requirements. Furthermore, requirements common to the entire product line should be written with variation points that can be filled in to create product specific requirements [Cle01].

Applying this on use case modeling, results in a two-layer use case model as shown in Figure 6. The top layer, the "Product Family Layer", holds the "Product Family Use Case Model" which consists of a number of product family use cases and a number of change cases. Product family use cases may be abstract, that is, written with variation points that require instantiation in product projects. Product family use cases may also be concrete, that is, not contain any variation points. The change cases of the product family layer hold impact links to product family use cases and may be used as described in Section 4.3.

The bottom layer, the "Product Layer", holds the "Product Use Case Model". A product use case model is created for each new product built within the family and it consists of the following three types of artifacts:

1. References to unchanged concrete family use cases that do not need instantiation.
2. Instantiated abstract family use cases.
3. New product specific use cases that are modeled as change cases.

This means that the product use case model only contains concrete artifacts, as would also be the case in traditional single system development.

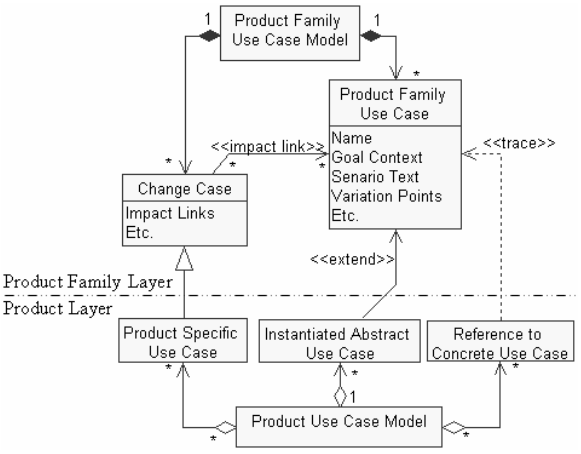

Figure 6: A two-layer use case model for product line development in UML

A problem with this approach is however that use cases provide the high level view of the product family, which leads to the problems mentioned in Section 4.1.

5 Our Approach: Marrying Feature and Use Case Modeling

We have described problems with existing approaches for variability management in use case models. Identified problems relate to lack of a high level view of the system family, poor support for traceability of variant behavior to the system architecture and a failure to recognize the difference between local and global variation points in the domain model.

Our approach to address these issues is to integrate feature and use case modeling. Similar to the work by Griss et al. [Gri98], we argue that feature models are better suited for domain modeling than UML use case diagrams and that a feature model therefore should be used as the high level view of the product family. We also propose that this feature model is used to group use cases into valid configurations based on what features they realize. Furthermore, we propose an approach for describing variant behavior in use case descriptions that better addresses the issues mentioned above.

5.1 Connecting Features and Use Cases to Cope with Variability

The mapping between features and use cases is a many-to-many relationship. One feature may be described by several use cases and one use case may be included in several features.

Connecting the feature model with the use case model and using the feature model as the high level view of the product family removes the need for the "Product Family Use Case Model" shown in Figure 6. The feature model is used as the domain model in the "Product family layer" and features are directly related to use cases. This results in a coherent product line requirements model as shown in Figure 7.

In the "Product layer", the "Product use case model" still represents the requirements of the system to be built within product family. Product specific functional requirements are modeled as product specific use cases. These product specific use cases are in turn modeled as change cases to form the basis for change impact analysis and cost estimates.

This model has two main advantages over the model shown in Figure 6. First of all, features and use cases are no longer stored in different models; this fact reduces the risk of inconsistencies between the models. Secondly, use case diagrams are no longer needed to provide the high level view of the product family;

instead features are used to describe the commonalities and variabilities of the domain.

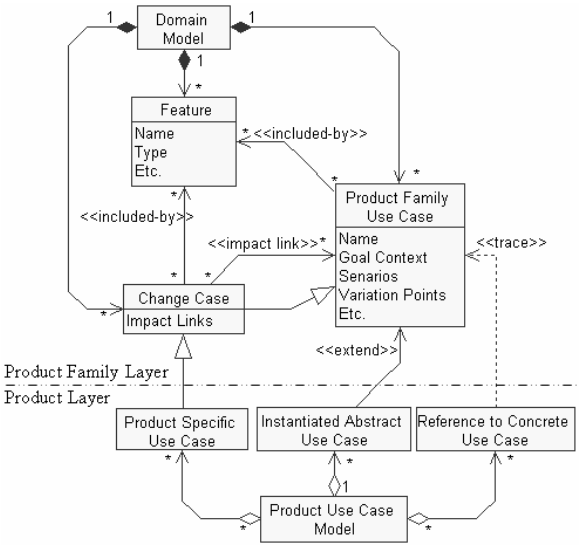

Figure 7: An overview of the proposed model in UML

5.2 Modeling Variability in Use Case Descriptions with Parameters and an Extended Version of the Black Box Flow of Events

We propose that use case scenarios be described using an extended version of the tabular RUP SE Black box flow of events notation shown in Figure 3. The extension provides the means to specify variant behavior using a very compact notation. The step identifier of the flow, which in the original notation is a simple running number, can be extended to specify variant behavior as follows:

- A number step identifier, such as “1”, identifies a mandatory step in the scenario as it does in the original notation.
- Several steps identified with the same number, such as “4”, “4” and “4”; identify a number of alternatives for one mandatory step in the scenario out of which exactly one must be selected.
- Several step identifiers with the same number and a consecutive letter, such as “4a”, “4b”, and “4c”, identify a number of alternatives for one mandatory step in the scenario out of which at least one must be selected.
- A number step identifier within parenthesis, such as “(2)”, identifies an optional step in the scenario.
- Several step identifiers with the same number within parenthesis and a consecutive letter, such as “(3)a”, “(3)b” and “(3)c”; identify a number of alternatives for one optional step in the scenario out of which at least one may be selected.
- Several step identifiers with the same number within parenthesis, such as “(3)”, “(3)” and “(3)”; identify a number of alternatives for one optional step in the scenario out of which exactly one may be selected.

Using an extended version or the black box flow of events notation enables application of the use case flow down activity described in Section 3.3, and thereby addresses the issue of poor support for tracing variant behavior to the system architecture.

As mentioned in Section 4.2, Jacobson et al. introduced the concept of parameterized use cases as part of the Reuse Driven Software Engineering Business (RSEB) in [Jac97]. We find parameters to be useful when defining reusable use cases, however not in the form presented in RSEB. Mannion et al. distinguished between local parameters, which have a scope exclusive to the single requirement that contains the parameter, and global parameters that have a scope of the whole set of family requirements, in their work on reusable natural language requirements [Man00]. We find this distinction to be very useful also when working with use cases; we therefore propose that this distinction be made. In our approach, the scope of a local parameter is the use case in which it resides and the scope of a global parameter is the whole domain model. Mannion et al. use the symbol ‘$’ to denote a local parameters and the symbol ‘@’ to denote a global parameter in requirements text, we have also adopted this notation.

Figure 8 shows an example of a use case scenario description using the proposed notation. The example use case is taken from an Automatic Teller Machine that utilizes the personal identification system described by the feature model shown in Figure 1. Steps “1”, ”4” and ”5” describe mandatory behavior, step “(2)” describe optional behavior and the consecutive steps identified with a “(3)” describe optional behavior out of which exactly one can be selected in the example. The use case scenario of the example does also hold two parameters, the global parameter @PIN_SIZE and the local parameter $MAX_CHIP_CARD.

Step	Actor Action	Black Box System Action	Black Box Budget Requirements
1	The use case begins when the **Customer** inserts the chip card into the ATM.	The **System** request that the customer should be identified.	Total response time is `$MAX_CHIP_CARD` *s*
(2)	The **Customer** presents its `@PIN_SIZE` character long PIN.	The System verifies the PIN.	Total response time is 0.1*s*
(3)	The **Customer** presents the fingerprint to the ATM.	The **System** verifies the fingerprint.	Total response time is 0.1*s*
(3)	The **Customer** presents the voice sample to the ATM.	The **System** verifies the voice sample.	Total response time is 0.1*s*
4	The **Customer** selects the amount of money to be withdrawn.	The **System** deducts the amount from the customer account and dispenses the cash to the Customer.	
5	The **Customer** takes the cash from the money dispenser.	The **System** is put into idle state.	

Figure 8: An ATM use case scenario description in the proposed notation

The example of Figure 8 shows small-grained variant behavior. It is however, also possible to specify whole scenarios, or even whole use cases as mentioned above, as being variants depending on the current modeling needs.

5.3 A Proposed Meta-Model

A meta-model for integration of features and use cases is provided to further improve the utility of the proposed approach. The meta-model, which is presented in Figure 9, is based on the Nokia feature meta-model shown in Figure 2. We have, however, focused on the relationship between features and use cases in our meta-model and therefore omitted some details about inter-feature and inter-use case relations for the purpose of readability.

The meta-model describes how global use case parameters, use cases and features are part of the domain model. Global use case parameters can be referenced by several use cases within a domain model. The meta-model also show how a use case can hold a number of local parameters. We also introduced a relation “instantiated-by” in our model. The purpose of the “instantiated-by” relation, is to indicate that parameters can be instantiated to the value of a feature property when a feature is selected to be included in a product.

The relation “included-by” is introduced in the meta-model to relate features to use cases and use case elements in following way:

- Use cases and features have a many-many relationship. Several features can include the same use case and one feature may include several use cases.
- Use case scenarios can be selected to be included by optional, one-out-of-many (alternative) or at-least-one-out-of-many features, only if its parent use case also is selected to be included in the system.
- Scenario steps can be selected to be included by an optional, one-out-of-many (alternative) or at-least-one-out-of-many features, only if its parent use case scenario also is selected to be included in the system.

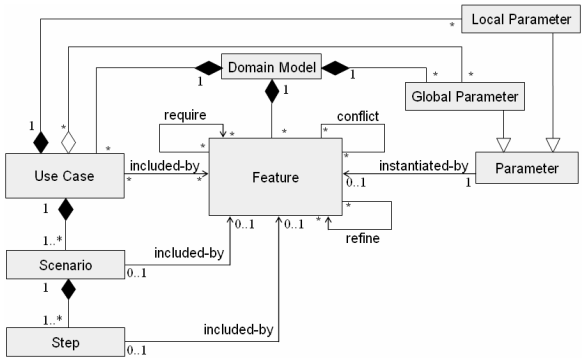

Figure 9: The Proposed Meta-model in UML

6. Applying the Approach

The product line requirements modeling approach proposed in this paper can be applied using standard office tools such as word processors and spreadsheets in combination with a basic UML modeling tool. Figure 10 shows an example of partial domain model describing a family of combat vehicle information systems where the feature “Diagnostics” are further described by the use case “Run Diagnostics” and the change case “Run Remote Diagnostics”. The example model was created using a commercial UML modeling tool by utilizing the UML stereotype mechanism.

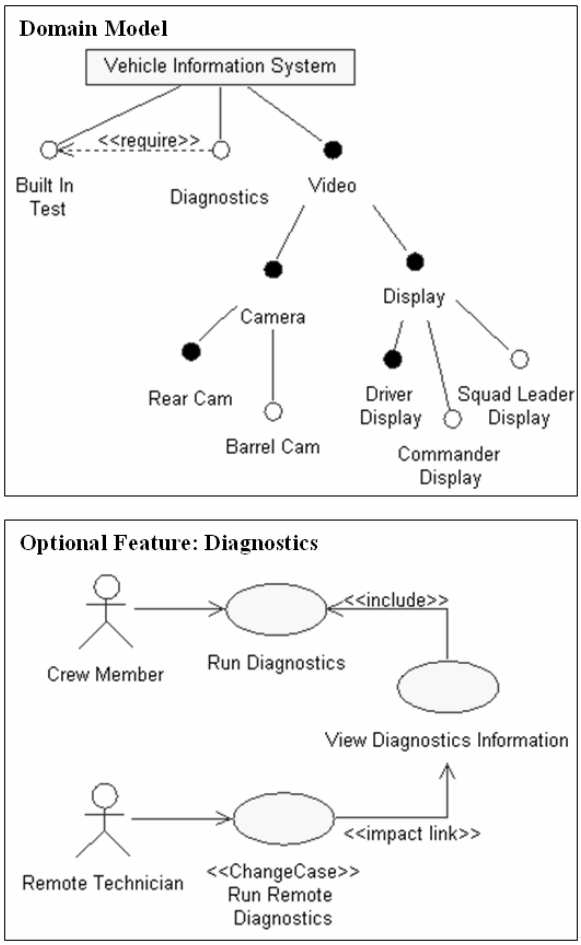

Figure 10: An example of a partial product family requirements model created using a commercial UML modeling tool.

6.1 Evaluation

The presented approach has been applied to representative examples of our domain, however not yet to full size development projects. Initial results indicate that the approach produces models well suited for our needs. However, experience has shown that for this type of product line requirements modeling to become efficient in every day practice, adequate tool support is an important factor. Unfortunately, today, product line development tools are almost nonexistent [Cle01]. There are some experimental tools and very few commercial tools available. However, these existing tools are only providing a subset of the functionality that we would like to see in a product line requirements engineering tool. Examples of the functionality we would like to see include baselineing of two-layer models as described in Section 4.4 for automatic generation of product use case models based on feature selections and consistency checking of product use case models based on feature relations.

The working group for "Practices and Issues Related to Product Line Tool Support" of the Fourth Product Line Practice Workshop identified a number of risks associated with using tools designed for single system development in product line development [Bas00]. The following list summarizes some of those risks:

1. There is a high risk that resources may be expended unnecessary because of lack of adequate tool support.
2. There is a high risk that artifacts become inconsistent because tools do not enable linkage between the different artifacts.
3. There is a high risk that tools become difficult to maintain because of the need for local modifications to enable product line development.

These risks do well describe our experience working with tools designed for single system development as well.

7. Summary and Conclusions

7.1 Summary

We have described how features, use cases and change cases can be integrated into a coherent two-layer product line requirements model.

The product family layer of the model is structured around a feature model that describes the capabilities of the product line. The features of this feature model are then further described by a number of abstract and concrete product family use cases. Furthermore, change cases that model possible future functional requirements of the family are also included in the family layer to aid architects in specifying a robust future safe architecture and to form the basis for change impact analysis when considering implementing new requirements.

The product layer of this model consists of the product use case model. The product use case model holds three types of requirements artifacts:

1. References to concrete family use cases
2. Instantiated abstract family use cases
3. New product specific use cases that are modeled as change cases.

This means that product use case model only hold concrete requirements artifacts as would also be the case in traditional single system development.

We have also described stronger means, than those earlier described in the literature, to define variant behavior within use case descriptions and mechanisms to trace those variants to the system architecture.

7.2 Conclusions and Future Work

Applying the approach on examples representative of our domain has resulted in artifacts well suited for our needs. However, lack of adequate tool support presents problems in our every day development efforts. Today, tool support for product line development is almost nonexistent. We therefore hope that our work will be enlightening to the tool vendor community.

The next step of our work is to apply the presented modeling approach to full scale development projects to validate its scalability. As part of this work, we will also consider implementation of some sort of experimental tool support to ease our development efforts.

References

[Bas00] L. Bass, P. Clements, P. Donohoe, J. McGregor and L. Nortrop, "Fourth Product Line Practice Workshop Report", Technical Report CMU/SEI-2000-TR-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000

[Cha01] G. Chastek., P. Donohoe. and K. Kang, "Product Line Analysis: A Practical Introduction", Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001.

[Cle01] P. Clements and L. Northrop, "Software Product Lines, Practices and Patterns", Addison-Wesley, 2001.

[Coc03] A. Cockburn, "Patterns for effective Use Cases", Addison-Wesley, 2003.

[Dor97] M. Dorfman and R. H. Thayer, "Software Requirements Engineering", Los Alamitos, CA, IEEE Computer Society Press, 1997

[Eck96] E. Ecklund, L. Delcambre and M. Freiling, "Change Cases - Use Cases that Identify Future Requirements", Proceedings of OOPSLA 96, San Jose, Ca, October 6-10 1996, pp.342-358.

[Fey02] D. Fey, R. Fajta and A. Boros, "Feature Modeling: A Meta-model to enhance Usability and Usefulness", Proceedings of the International Conference on Software Product Lines (SPLC2), August, 2002, pp. 198-216.

[Gri98] M. Griss, J. Favaro and M. d'Alessandro "Integrating Feature Modeling with the RSEB", Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June 2-5 1998, pp. 76-85.

[Jac97] I. Jacobson, M. Griss and P. Jonsson, "Software Reuse – Architecture, Process and Organization for Business success", Addison-Wesley, 1997.

[Joh02] I. John and D. Muthig, "Tailoring Use Cases for Product Line Modeling", Proceedings of the International Workshop on Requirements Engineering for Product Lines, 2002, pp. 26-32.

[Kan90] K. Kang et al, "Feature Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Kan98] K. Kang, S. Kim, J. Lee, E. Shin and M. Huh, "A Feature Oriented Reuse Method with Domain-Specific Reference Architectures", Annals of Software Engineering 5, 5 September 1998, pp.143-168.

[Kru00] P. Kruchten, "The Rational Unified Process, An Introduction", Second Edition, Addison-Wesley, 2000.

[Man00] M. Mannion, O. Lewis, H. Kaindl, G. Montroni and J. Wheadon "Representing Requirements on Generic Software in an Application Family Model", Proceedings of the International Conference on Software Reuse (ICSR-6), 2000, pp. 153-196.

[Maβ02] T. Von der Maβen and H. Lichter, "Modeling Variability by UML Use Case Diagrams", Proceedings of the International Workshop on Requirements Engineering for Product Lines, 2002, pp. 19-25.

[RUP03] "The Rational Unified Process for Systems Engineering Whitepaper", Version 1.1, Available at: http://www.rational.com/media/whitepapers/TP165.pdf, October 2003

[Tri03] J.C. Trigaux and P. Heymans, "Modelling variability requirements in Software Product Lines: A comparative survey", Technical report PLENTY project, Institut d'Informatique FUNDP, Namur, Belgium, november 2003

[UML03] "OMG - Unified Modeling Language", Version 1.5, Available at: http://www.uml.org, March 2003