The Hyper/UML Approach for Feature Based Software Design

Ilka Philippow
Technical University of Ilmenau
PSF 100565
98684 Ilmenau, Germany
Ilka.Philippow@tu-ilmenau.de

Matthias Riebisch
Technical University of Ilmenau
PSF 100565
98684 Ilmenau, Germany
Matthias.Riebisch@tu-imenau.de

Kai Boellert
Bonndata (Zurich Group)
Riehler Str. 90
50668 Koeln Germany
Kai.Boellert@zurich.com

ABSTRACT

The market requests complex but adaptable software systems. There are different concepts to meet this demand, for example software reusability, component-based development, agile processes, generative programming and domain analysis. For similar products within a domain product lines are a very promising approach for shortening development time and cost and for improving quality. Software product lines combine some of the mentioned concepts. In a software product line there is a common core for all target products and variable components for building different products. Using product lines new products are created based on customer-ordered features. The feature driven development of software systems becomes more and more important not only for product lines but also for the better maintainability of software systems. Implementing variability by composition enables a partly automated development process of products. Extending composition techniques from source code to models and design support large systems and their evolution. In this paper the concept of Hyper/UML as UML extension is introduced for supporting the feature oriented separation of concerns, the feature driven design and the feature based automated generation of software products. The concept is based on the Hyperspace approach.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE)—*Object-oriented design methods*; D.2.13 [**Reusable Software**]: Domain engineering

General Terms

Design, Languages

Keywords

Hyperspace, Software Product Lines, Feature Model

1. INTRODUCTION

The market requests software systems that are changeable faster and in a more complex way than ever before. For software development, aspects of the reuse of proven former development models, decisions and components become more and more important for cost reduction, lowering of development time and increasing software quality. Reuse can be applied at different levels. It is possible to reuse source code as modules, functions, classes and components, or on a higher abstraction level by reusing of former developed artifacts of analysis, design, and architecture. Frameworks and product lines allow the reuse of both source code and models of analysis and design. Software product lines can be considered as a fundamental approach that is connected with expectations for enhancements in reusability, adaptability, flexibility, and control of complexity and performance of software and software development processes.

By software product lines a "group of similar products" out of a specific problem domain is described that are based on a system family architecture offering a "common set of core assets" [2] and variable parts. Variable parts can be changed or adapted to satisfy the special needs of an application. The customer orders a particular product within the family based on a set of offered product features. Each feature represents a set of requirements. The production can be carried out automatically by applying generative techniques and product generators. In figure 1 the relations between the development process of product lines and product line based application development is shown, represented by a so-called six pack model [7] of activities of the iterative processes. The development of product lines (see the upper three blocks) and the development of an application (see the lower three blocks) are connected by the families requirements model, the families reference architecture and the implemented components. The left block of activities consists of domain engineering activities that lead to feature models based on old and new occurring requirements. The requirements serve for the definition of the core and variable parts. The activities of the middle block are focused on the modeling of core and variable elements and the architecture design. The left activity block implements and tests in the necessary system components.

In product line's commonality and variability can be repre-

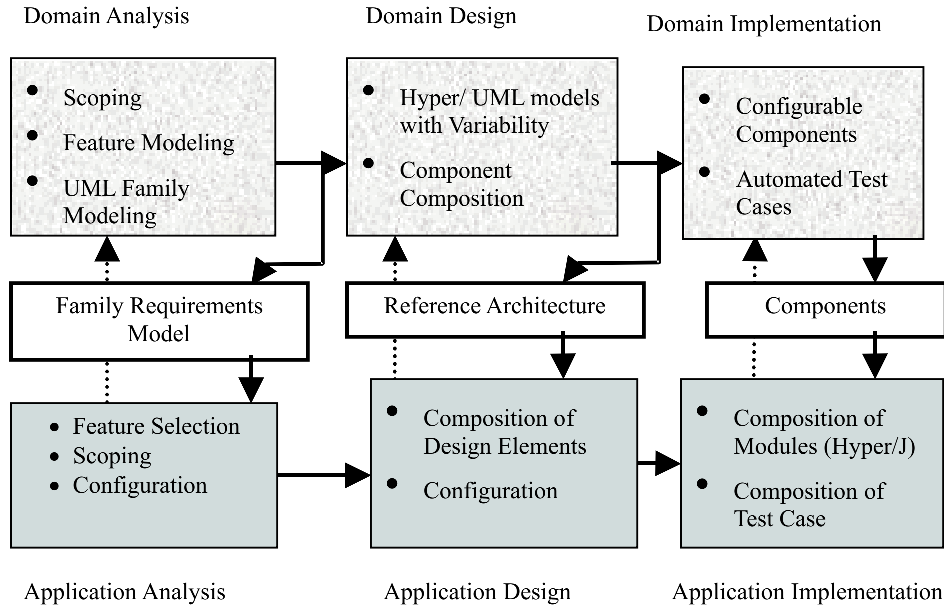

Figure 1: Relations between the Development of Product Lines and Product Line Based Application Development

sented by feature models [4]. A feature represents a property of a product that is valuable for a customer to distinct between products of the product line. A feature model contains all features covered by a product line with their dependencies and their variability. Based on feature models costumers configure their wanted system and order it. For the feature based automated product generation the components of a product line have to be developed in a special way. Variable components must be able to match a variety of interfaces. The prefabricated components can be combined to build new products quickly. This is an advantage compared to adaptation techniques of other product line approaches. The Hyperspace approach is suitable for decomposing and composing of software not only by classes and objects, but simultaneously by features. By now its implementation is carried out for Java (Hyper/J). In this paper the Hyperspace approach is integrated into the Unified Modeling Language (UML). The defined and introduced so called Hyper/UML enables composing and decomposing of components on model level to support the development and evolution of larger product lines and their architectures. It enables the automated generation of software products.

2. THE HYPER SPACE APPROACH

The Hyperspace Approach [6] is a generic technique focussed on the decomposition and composing of software systems according to concerns. The Hyperspace separates and describes system properties using a multidimensional matrix (figure 2). Relevant concerns of a concern type that are identified during system decomposition have to be represented using a particular dimension. For object oriented software systems, especially for feature For object oriented software systems, especially for feature driven software development typically two dimensions are relevant: for classes and for features. Features are relevant for concerns because the later composition is driven by features.

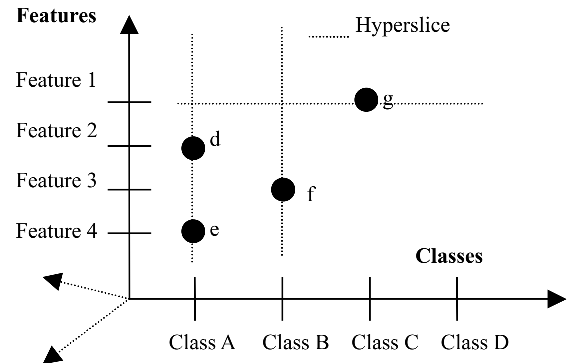

Figure 2: Hyperspace matrix with two relevant dimensions, classes and features

Simultaneously, the system decomposition leads to different elements: classes, use cases, activities, actors, etc. There are primitive atomic and compound elements. All elements are mapped onto exactly one point in the Hyperspace matrix according to which concern a unit belongs to.

The *Hyperspace* is used for representing relevant concerns in dimensions and for the assignment of system elements to concerns. Elements from one or more concerns can be encapsulated in the so called *Hyperslice*. One Hyperslice contains all elements relevant for one concern, even if they are scattered over a system. In Hyper/J the implementation of *Hyperslices* is done by using packages. A very important property of *Hyperslices* is their declarative completeness. This is a prerequisite for the mutual independence of Hyperslices by

avoiding of overlapping elements. Independence of Hyperslices is necessary to carry out composition automatically. To compose a particular system by concerns - in fact features - all Hyperslices related to the relevant concerns are composed. The composing process leads to a *Hypermodule* as a simple union of the Hyperslices. For an automated composition integration relationships and rules for a particular Hypermodule have to be determined by the developer. For the simplest case that all elements are orthogonal special integration relations are not needed. To get executable *Hypermodules* all declarations have to be implemented.

3. HYPER/UML FOR FEATURE ORIENTED AND MODEL-BASED SYSTEM DEVELOPMENT

In the past, the Hyperspace approach was implemented for the configuration Hypermodules on code level in Java by Hyper/J [6]. For designing and modelling commonality and variability on model level, for instance using the UML the Hyper/UML was developed in [1] and is introduced in this paper.

Hyper/UML was designed as part of a methodology for model-based development of product lines called Alexandria [12]. In this methodology, feature models are applied for structuring user requirements and for designating common and variable features of a product line. Common features will be implemented in the product line's common core, whereas variable features will become variable components. By applying the Hyperspace approach here, these components are built as Hyperslices by applying features as concerns. This way, 1-to-1 relationship of features and components are facilitated and fine-grained components are built. A component is modelled using Hyper/UML and is implemented in Hyper/J.

For developing a family of software systems, a product line provides a feature model, a set of design model components - in fact Hyperslices built using Hyper/UML - and implementation components based on these design components - in fact Hyper/J Hyperslices. A new software system is derived

- by selecting a valid set of variable features
- by composing the relevant Hyper/UML Hyperslices to a design model of the system
- by composing the relevant Hyper/J Hyperslices to the system's code

Any additions of design model and code are put into an additional Hyperslice that is composed with the other Hyperslices. As a consequence, the additions are separated and therefore they are usable for later systems as well.

3.1 Hyper/UML Description

Hyper/UML is an application of the Hyperspace approach for the UML 1.4. It is an extension of the UML. At the beginning there is an empty package (figure 3, part (a)) that has to be filled step by step. Therefore it is necessary to answer the following questions:

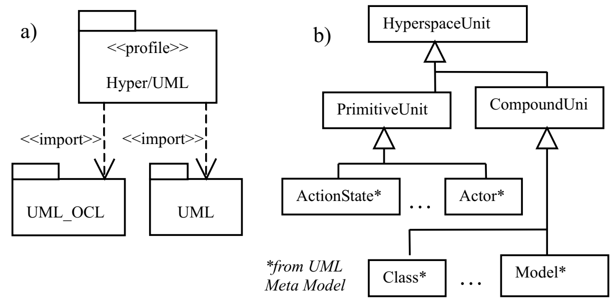

Figure 3: Hyper/UML Profile and Elements

- *Elements*: Which are the relevant elements? Are the elements primitive (atomic) or compound ones?
- *Hyperspace*: How is a hyperspace to be described? Which is the way to assign elements to the identified concerns and dimensions?
- *Hyperslice*: How is a Hyperslice to be encapsulated and described? Which is the way to declare elements that occur in different slices?
- *Hypermodule*: How is a Hypermodule to be described? Which are the necessary integration relations?
- *Integration relations*: What integration relations are expected? Are there pre and post conditions?
- *Tools*: Which are the requirements for the automated application of integration rules during hypermodule configuration?

The assignment of UML elements as subclasses of the abstract Hyper/UML meta model class HyperspaceUnit is shown in figure 3 Teil b. The concepts of the Hyperspace are mapped onto nearly all model elements of the UML according to [5] with the exception of special elements for component and deployment diagrams. A distinction between atomic (PrimitveUnit) and composed (CompoundUnit) elements is necessary for a later composition. Atomic elements are actors, associations, attributes, operations, signals, and states. The other elements like activity graph, class, interface, state machine and use case, are composed and contain different elements.

The integration and description of Hyperspace and Hyperslice is presented in Figure 4. A Hyperspace is associated with dimensions that are separated into concerns. Additionally a meta class NoneConcern belongs to each dimension. Hyper/UML elements are indirect instances of HyperspaceUnits and assigned to Hyperspace concerns. If there is no assignment for an element in accordance to the Hyperspace approach it is by default assigned to NoneConcern.

Hyperslices can be defined as a subtype of UML package (figure 4) for encapsulation of model elements assigned to one concern. There is no part-of relation between Hyperslices because they must be independent from each other. Hyperslices can be composed to a particular software system by using a Hypermodule.

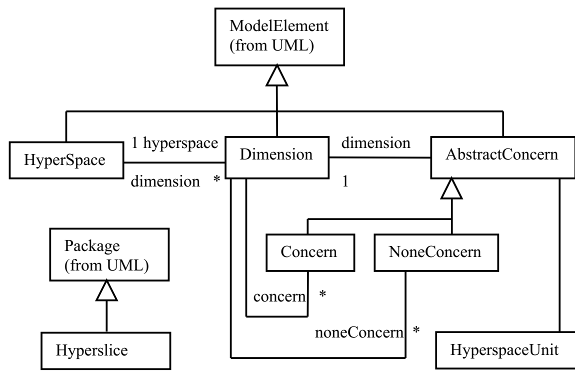

Figure 4: Hyperspace and Hyperslice within the Hyper/UML

A Hypermodule is considered as a special Hyperslice with associated integration relationships for the composition (figure 5). Integration relationships determine the steps and conditions for the composition of Hypermodules. The here explained integration relationships are based on those that exist in Hyper/J and the experience and recognized necessities achieved during a case study project. The case study (80 use cases, 300 classes) and the integration relationships are described in detail in [1]. The case study covers the most frequent integration situations. For the future, additional integration relationships are planned, e.g. for runtime variability. For the here introduced Hyper/UML defined four relationships were defined:

- *Merge*: This is the main integration relationship. It serves for the composing of all semantic identical elements. The very simple basic principle is to bring together different elements and to merge identical elements.
- *Summary*: This relationship is applied for determining of the return parameter in case of operation merging.
- *Order*: This relationship serves for the determination of sequences for active elements like operations or use cases in order to merge these elements of the same type.
- *Override*: This relationship enables the complete replacement of one element by another element of the same type.

Like all complex modelling processes Hyper/UML modelling has to be supported by a tool that is expected to offer the following services and functions:

- *Hyperspace Management*: Based on the user definition the *Hyperspace* with its dimension and the mapping of elements to *Hyperspace* points has to be established.
- *Hyperslice Modelling*: This function is similar to UML modelling and easy to realize. A new notation is not necessary, only some extensions for validating the *Hyperslice* correctness.

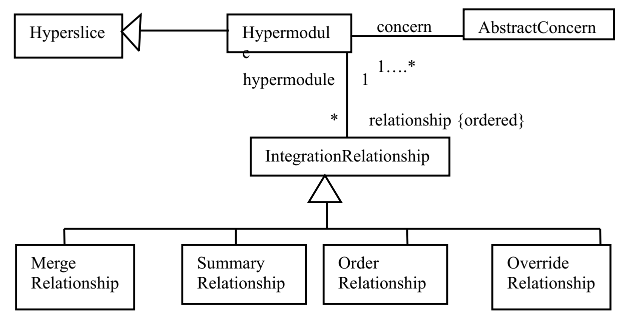

Figure 5: Hyperspace and Hyperslice within the Hyper/UML

- *Hypermodule Management*: For the management of *Hypermodules* system developers have to provide a specification. The specification must include the concerns that have to be integrated and the concrete integration relations. The specification can be carried out textual. A special graphical notation is not really relevant. But for maintenance reasons it is necessary that the tool presents the integration results graphical by UML diagrams. The problem is the realization of an algorithm for the automated re-ordering of elements in diagrams.

3.2 Integration Relations

Integration Relations enable the automated composition of *Hypermoduls* in accordance to the desired concerns. All elements assigned directly or indirectly to one of the relevant concerns have to be included in the composition. The composition process leads to a*Hypermodel* package in two steps:

- Relevant Hyperslices and their elements are copied into the *Hypermodul*
- Integration relations are processed.

In [1] a complete semantic description for Hyper/UML and the four integration relationships is specified for all applicable UML diagram elements. They are described by rules and rule related operations using UML's Object Constraint Language OCL. The OCL description is necessary for an automated check of various pre and post conditions. For staying in the limited scope for this paper it is possible to introduce only the general ideas and results.

The main integration relation *merge* is focussed on particular elements that have to be combined. Elements have to be distinguished between target elements (targetUnit) that contain the merging result and source elements (figure 6). Copies of source elements have to be removed from the Hypermodule after merging. For the merging of elements there are determined basic rules:

- target elements must not be source elements at the same time
- only elements from the same type can be merged

- the Hyper/UML element types that can be merged are: actor, attribute, class, Hyperslice, interface, model, operation, package, use case, stateMachine, simpleState, compositeState, signal, activityGraph, actionState, subactivityState

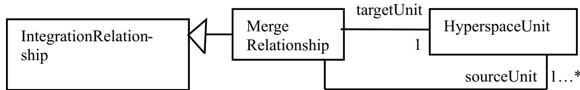

Figure 6: Integration Relation Merge

The fundamental principle of merging is that corresponding equal elements have to be merged to one collecting element. For special elements this basic rule can be replaced by a concrete override relationship. Hyper/UML defines for all admitted elements rules, pre and post conditions for their merging. As an example in this paper the pre and post conditions for the merging of classes are given in Table 1. Attributes, Associations and Operations have to be handled connected with the particular classes because they are referenced by other model elements indirectly through their class. The very interesting aspect is how to merge attribute values, associations and other dependencies and constraints. The relevant pre and post conditions are summarized in Table 2. During the merging process several exceptions can arise that lead to an interrupt of the automated merging process.

The merging of operations can be combined and extended with an integration relation *order* and/or *summary.* Target operations contain implementations if they exist. The sequence of the implementation execution is undefined or has to be determined by a *Hypermodule order* integration relation. The sequence of implementations influences the parameter. The first implementation gets the parameter from the operation call. Every next implementation gets parameter left from the previous implementation. The last implementation returns values to the initiator of an operation call. To modify this management of return parameters a *Hypermodule summary* integration relation can be used.

The integration relation *order* is associated with the elements that have to be ordered (figure 7). By the *order* relation the merging sequence for elements can be determined.

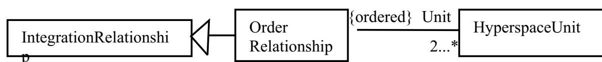

Figure 7: Integration Relation Order

The Hyper/UML element types that can be ordered usefully and semantically relevant are operations, use cases, activityGraphs, activiStates and subactivityStates. The semantic of the integration relation *order* differs depending on the element type:

- If operations have been merged than the target operation processed the implemented operations in the order given by the relation order rule.

Table 1: Merging of Classes

Pre Condition:
• all Classes have to be either abstract or concrete • the target class inherits from parents of source classes, sub classes of source classes inherit from the target class
Post Condition:
• the target class has received in addition to their own all attributes, associations and operations of the involved source elements • redundant inheritance relations must be removed • the target class receives in addition state machines of source classes, state machines have to be merged, too • instances of classes in UML can be found in interaction and activity diagrams; after merging these instances belong to the parent class
Exceptions:
• in UML cycles in inheritance relations are not allowed • using attributes and associations in subclasses with the same name like in the parent class is not allowed in UML • interface classes do not contain implementations of all needed operations

- If use cases have been merged than the target use case description contains sequence related order in accordance with the relation order rule. The use case order has to be transferred to involved activity graphs.
- If activity graphs have been merged the sequences of graphs within the target activity graph corresponds with the relation order rule.
- If activity states have been merged the single states are connected in the given order.

In case of operation merging the integration relation *summary* describes the way how to determine the parameter value that have to be returned to the initiator of an operation call (figure 8). SummarizedUnit contains the operation with the merged implementations. SummaryFunction realizes the return value. The merged operations and summaryFunction are concrete operations with return parameter of same type. SummaryFunction is a class function. It possesses an in- parameter of identical type with the relevant return parameter. Type identity has to be proved considering all integration relations of the Hypermodule by using

Table 2: Merging of Attributes, Associations and Operations

Pre Condition: • an attribute of the same name must not possess different values • attributes must be of the same type • all attributes have to be either class attributes or instance attributes with the same changeability, initial value, and multiplicity • all operations must be either abstract or concrete and of the same concurrency • all operations have to be either class operations or instance operations with the same structure and default values of their parameter list • associations must refer to identically types • all associations have to be general, or they define an aggregation or a composition • for merging of associations navigate ability changeability, multiplicity must be the same
Post Condition: • target elements keep names if there are given names • element related constrains from source elements are assigned to the target element • target elements depend on model elements that contain source elements, source elements are associated to target elements; all redundant dependencies and association have to be removed • target attributes get either the given initial values from the source or none value • parameters of target operations get either given default values or none values • the complete result of operation merging depends on predefined and processed *order* and *summary* relationship • operation calls are directed to target operations; in a running system on model level an operation call takes place as message interaction between objects or as entry-, exit-action, state- or transition activity in state machines within the target state machines
Exceptions: • constraint related inconsistencies lead to an interrupt, for an automated inconsistency check • constrains have to be formulated using a formal language like OCL

OCL description. The summarizedUnit collects the return values of the merged implementations and calls the summaryFunction. SummaryFunction generates the return value that is returned to the operation initiator by the summarizedUnit.

The merging of operations can be combined and extended with an integration relation *order* and/or *summary*. Target operations contain implementations if they exist. The sequence of the implementation execution is undefined or has to be determined by a *Hypermodule order* integration relation. The sequence of implementations influences the parameter. The first implementation gets the parameter from the operation call. Every next implementation gets parameter left from the previous implementation. The last implementation returns values to the initiator of an operation call. To modify this management of return parameters a *Hypermodule summary* integration relation can be used.

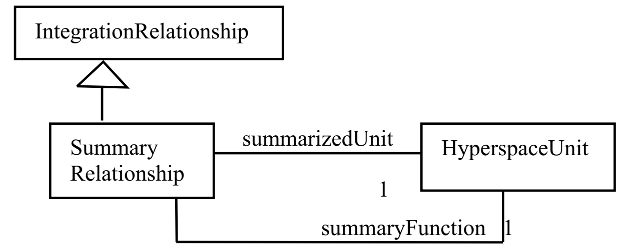

Figure 8: Summary Relationship

The integration relation *override* (Fig. 9) is used for the complete replacement of one element by another element of the same type. The integration relation override is associated with the element that has to be replaced (overriddenUnit) and the substituting element (overridingUnit). Substituted elements are removed from the Hypermodul. A later reference by another integration relation leads to an exception interrupt.Replacement requires the keeping of interface behaviour. Hyper/UML provides OCL constrains for checking the consistency within the Hypermodule for the following elements: operation, actionState, subactivity state. Table 3 shows the pre and post conditions for operations and activity states.

Hyper/UML elements and integration relationships are defined for all UML diagram elements except those of implementation and deployment diagrams. According to the Hyperspace approach

3.3 Feature-driven Software Composition Using Hyper/UML

Feature driven software development is provided for the development of reusable and adaptable software systems, e.g. product lines. For modeling variability, feature models can be used that are proposed for the method FeatuRSEB in [3]. Feature models represent common and variable parts by using mandatory (full circle), optional (empty circle) and alternative features. There are slightly modified in [8] to overcome some ambiguities. Fig. 10 shows partly an easy understandable feature model for a library product line. In addition to the common alternatives it is allowed to select

Table 3: Overriding of Operations and Activity Graphs

Operations
Pre Condition: Both, the substituting and the substituted operation are defined in an identical class. They possess same names and parameter lists with type identical parameters and equal default values, if given. Both operations are either abstract or concrete. **Post Condition:** The particular element is completely replaced. Operation calls are directed to the replaced operation
Activity Graphs
Pre Condition: Both, the substituting and the substituted activity state belong to an identical activity graph. This condition has to be proved considering all integration relations of the Hypermodule by using OCL description. **Post Condition:** The activity state is replaced completely including of its entry action, constrains and property values. Object flows and transitions are not changed. Redundant transitions have to be removed. Transition related monitoring conditions are AND connected. If there are inconsistencies between conditions the replacing process is interrupted. This has to be proved and carried out using OCL notations.

one or more features of a feature set, described by the given multiplicities similar to UML. That way a better accordance to an architecture description using UML class diagrams and the improvement of traceability can be achieved. The usual exclude and/or require relationships enable the expression of further dependencies very similar to UML-stereotypes. Inheritance relations like in UML class diagrams are visible. Simultaneously the feature derived classes have to be determined. During the next step the two dimensional Hyperspace for features and classes and the Hyperslices have to be established. In Fig. 11 a part of the Hyperspace is shown for the feature combination: *library core* and *overdue notification*. Each Hyperslice has to be fully declared and encapsulated within a package For notification by both, email and mail, the three Hypersclices Core, Notification by mail and e mail must be merged in a Hypermodule (see Fig. 12.).

If applying the FeatuRSB method, variation points like *overdue notification* are demonstrated by additionally use case models. As shown in the Fig. 13, overdue books can be modeled with two use-cases bound to a variation point. Depending on the selected features, the use-cases will be part of the derived application or will be left out. Use Cases are often refined by activity and state models.

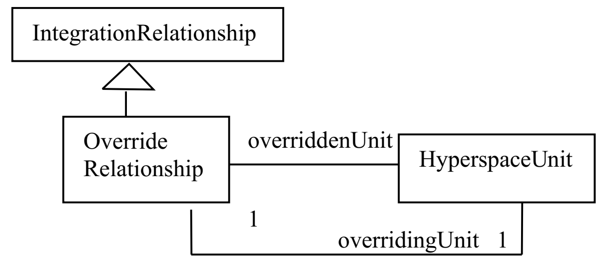

Figure 9: Integration Relation Override

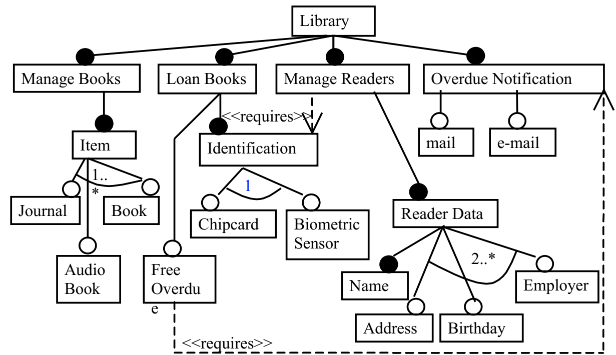

Figure 10: Feature Model of a Library Product Line

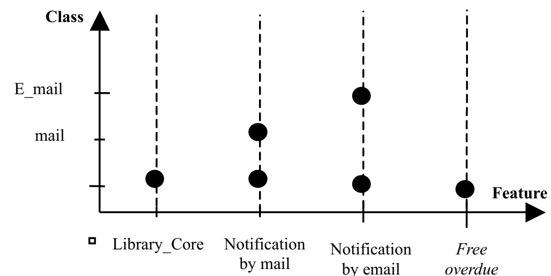

Figure 11: Library Product Line Hyperspace (partly)

In case there are readers e.g. the teaching staff of an university that do not have to pay at once a fee for overdoing the feature *Free Overdue* have to be included. This case is shown in Fig. 14 as example for the merging of state machines. Teaching staff members can keep books so long as possible if there is no contradicting order by other readers.

The design of feature-oriented components supports the composition of similar software products. The Hyper/UML can be integrated into the feature oriented method FetuRSEB [3]. For this case the briefly described development process is characterized by the following steps:

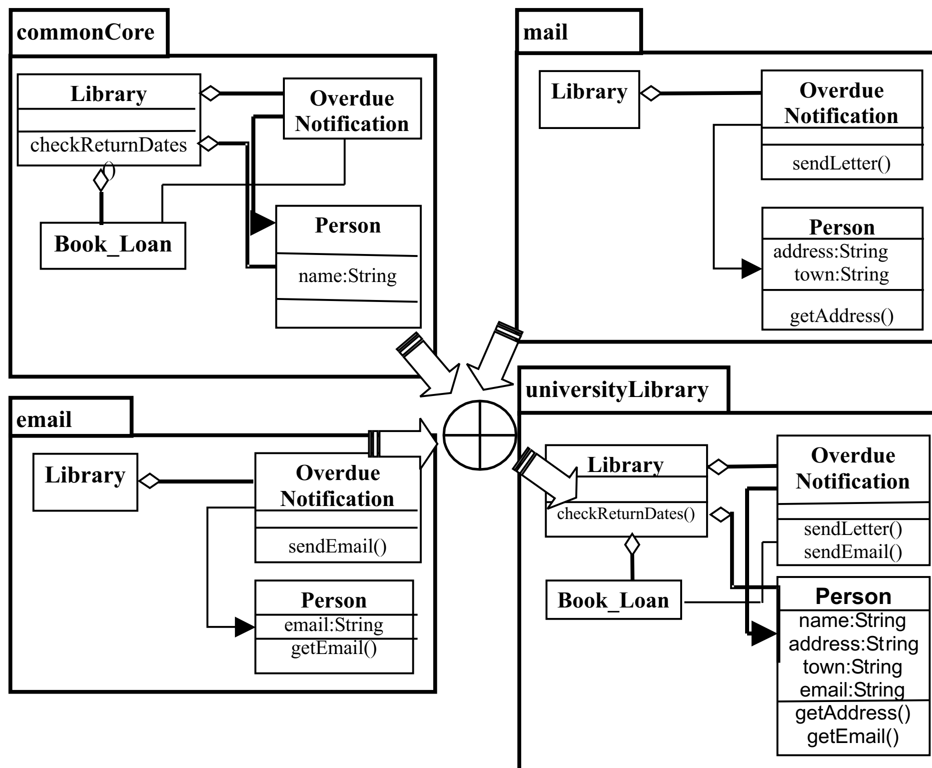

Figure 13: Merging of the Class Diagrams for Overdue Notification (partly)

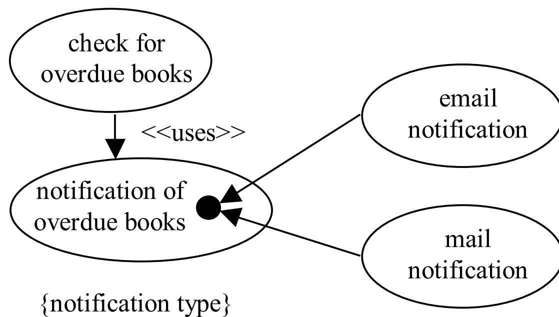

Figure 12: Use Case Modeling of Variability

- structuring and modeling commonality and variability using feature models
- decomposing and describing of features by use cases
- derivation of classes, use case refinement by activity or state models
- establishing of Hyperspace and Hyperslices
- defining the architecture and design of Hyperslices
- tracing and connecting of relationships and dependencies, and determining of integration dependencies for a later Hpermodule specification
- implementing in an object oriented programming language

The architecture consists of Hyperslice packages; each Hyperslice contains exactly use cases, activity and state models and classes of one feature. On model level the Hypermodule specification in Hyper/UML syntax. If Java is used, Hyper/J can be applied for Hyperslice implementation and the later composition of Hypermodules.

The development of architecture and design is performed in separate processes the Hyperspace and for each Hyperslice. The Hyperslice development processes are coordinated by the Hyperspace process especially concerning visibility, name space and interfaces.

4. RELATED WORKS

There are several works concerning Generative Programming and Aspect Oriented Programming, that are very similar to the Hyperspace approach used here. However, in our assessments the Hyperspace approach has offered better combination possibilities of components and less dependencies between them. The GenVoca approach [16] has shown stronger limitations for larger systems and for program comprehension. Most of the works in that area are based on programming languages, without the use of model-based composition. We found only the works of Clarke [11] with similar proposals, however without capabilities for composing behavioral models. There are some methodologies for the development of Software Product Lines, i.e. Bosch's method [20] and KobrA [15]. However, they implement variability using code abstractions and design patterns. As a consequence, the implementation of new software systems is to be done manually. Feature models are used by newer ap-

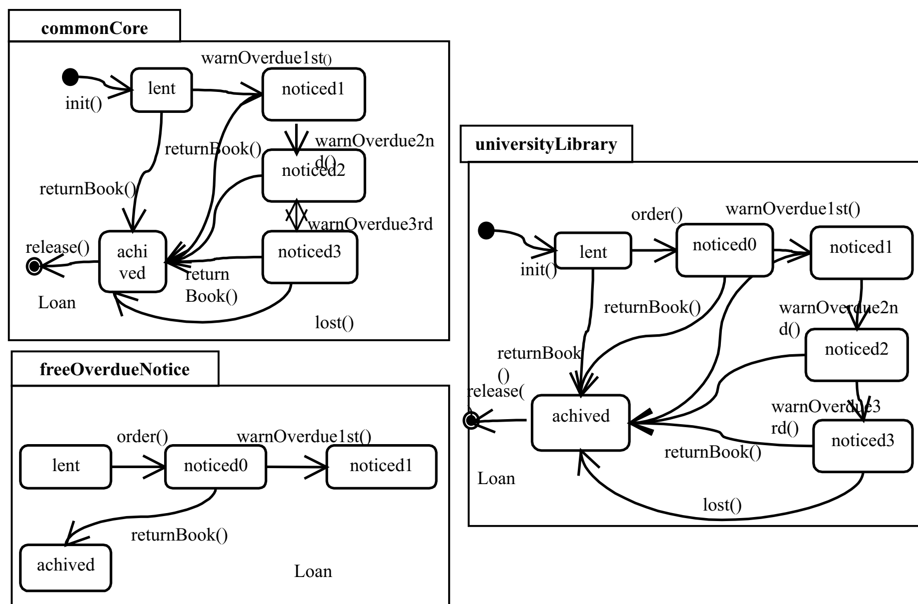

Figure 14: Merging of the state diagrams for including Free Overdue Notice concerning the class Loan

proaches than FODA, i.e. FORM [13] and FOPLE [14]. However, the definition of views and categories there is weak, the distinction between the different views and categories is not strong enough for an industrial application of the approach. Therefore, we developed a new definition of Feature models and extended it by a formal language for feature dependencies [19]. Mutual influences of features are subject of works about Feature Interaction, e.g. [17]. These methods have to be used to solve dependencies between features before Separation of Concerns is applicable. These methods provide an important addition to the work described here. Component architectures with pluggable components, e.g. according to the Eclipse approach, provide promising ideas for implementing variability for software product lines. Currently, there is a lack of methodical support for model-based development of larger systems. However, for the future this plug-in components constitute a very important issue, similar to the ideas of the Model-Driven Architecture MDA [18].

5. CONCLUSION

This paper has introduced the integration of the Hyperspace approach to UML for the feature driven decomposition and composition of similar software products. For applying the Hyperspace approach on model level, Hyper/UML was developed and explained in this paper. By Hyper/UML the Hyperspace concept is mapped onto UML and provides an UML extension especially for the development of product lines. By implementing variability using composition a higher degree of automation of product development is possible, compared to other implementation techniques. To enable tool support and automation, models and relations are partly defined by the Object Constraint Language OCL.

The approach has been applied in two case studies, described in detail in [1] and [10]. In the results it has been shown that the approach is very suitable for medium size product lines (e.g. 18.000 ELOC for about 42 features). In the case studies for Hypermodule specification on model level a self made tool has been used that passes the specification to the Hyper/J tool for composing Hyperslices to executable systems. Improved tool support on model level remains future work that can only be solved by or in cooperation with commercial tool provider.

A further aspect of interest is the testing of composed systems. Founded by the German Research Foundation DFG current research is focused on the automatic, model based test case generation. Test cases are composed similar to models and source code by using the Hyperspace approach.

6. REFERENCES

[1] Boellert, K.: Object-Oriented Development of Software Product Lines for Serial Production of Software Systems (in German: Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen). Doctoral Thesis. Ilmenau Technical University, Ilmenau, Germany. (2002)

[2] Clement, P.; Northrop, L.: A framework for software product line practice, version 2.7. (1999)

[3] Griss, M., Favaro, J., d'Allesandro, M.: Integrating Feature Modeling with RSEB. Hewlett-Packard. (1998)

[4] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA. (1990)

[5] OMG: Object Management Group. Unified Modeling Language Specification, Version 1.4. (2001)

[6] Ossher, H.; Tarr, P. Multi-Dimensional Separation of Concern and the Hyperspace Approach. In Software

Architectures and Component Technology. Kap. 10. Kluwer Academic Publishers (2001)

[7] Philippow, I.; Streitferdt, D.; Riebisch, M.: Reengineering of Architectures for Product Lines. European Conference on Object Oriented Programming (ECOOP 2003) Workshop on Modelling Variability of Object Oriented Product Line, Darmstadt (2003)

[8] Riebisch, M., Bllert, K., Streitferdt, D., Philippow, I.: Extending Feature Diagrams with UML Multiplicities. 6th World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, CA, USA; June 23 - 28, 2002. Society for Design and Process Science, Session 4, 1-7 (2002)

[9] Tarr, P., Ossher, H.: Hyper/J User and Installation Manual. In: Multi-Dimensional separation of Concerns: Software Engineering using Hyperspaces. www.research.ibm.com/hyperspace/ (2001)

[10] Halle, M.: Case Study for Developing Reusable Components in Software Product Lines (in German: Fallstudie zur Entwicklung wiederverwendbarer Komponenten im Rahmen von Software-Produktlinien). Diploma Thesis. Ilmenau Technical University, Ilmenau, Germany (2001)

[11] Clarke, Siobhan: Extending standard UML with model composition semantics. in Science of Computer Programming, Volume 44, Issue 1, pp. 71-100. Elsevier Science, July 2002.

[12] Riebisch, Matthias: Evolution and Composition of Software Systems - Software Product Lines as a Contribution to Flexibility and Longevity (in German: Evolution und Komposition von Softwaresystemen - Software-Produktlinien als Beitrag zu Flexibilitt und Langlebigkeit). Habilitation thesis. Ilmenau Technical University, Ilmenau, Germany (2003). Submitted.

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

[14] Kang, K.C.; Lee, K.; Lee, J.: FOPLE - Feature Oriented Product Line Software Engineering: Principles and Guidelines. Pohang University of Science and Technology, 2002.

[15] Atkinson, C., et al.: Component-based product line engineering with UML. Addison Wesley, 2002.

[16] Batory, Don und Bart J. Geraci, Bart J.: Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering (23)(2), Seiten 67-82 (February 1997).

[17] Calder, M.; Kolberg, M.; Magill, M.H.; Reiff-Marganiec, S.: Feature Interaction A Critical Review and Considered Forecast. Elsevier: Computer Networks, Volume 41/1, 2003. S. 115-141

[18] OMG Model Driven Architecture. Object Management Group. http://www.omg.org/mda/

[19] Riebisch, Matthias: Towards a More Precise Definition of Feature Models. In: M. Riebisch, J. O. Coplien, D, Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines. BookOnDemand Publ. Co., Norderstedt, 2003. pp. 64-76.

[20] Bosch, Jan : Design and Use of software architectures - adopting and evolving approach. Addison-Wesley, 2000.