Towards a General Model of Variability in Product Families

Martin Becker
System Software Group, University of Kaiserslautern
Kaiserslautern, Germany
mbecker@informatik.uni-kl.de

Abstract

The increasing amount of variability in software systems meanwhile leads to a situation where the complexity of variability management becomes a primary concern during software development. Whereas sound methodic support to analyze and specify variability on an abstract level is already available, the corresponding support on realization level is still lacking. The goal of this paper is to pave the way towards more systematic and consequently more efficient approaches to manage variability. To this end, it discusses the different motivations for variability in product families and the interrelationships between the specification and realization of variability. The paper further identifies appropriate concepts and interrelates them in form of a general model of variability in product families. In addition to this meta-model, the paper outlines an instantiation of the model: our language to specify variability in product family assets.

1. Introduction

During the past few years a noticeable shift towards an increased amount of variability[1] in software systems went through the software industry. The reasons for the increase of variability are twofold. First, variability has been recognized as the key to systematic and successful reuse. Especially in family-based approaches as software product lines or software product families, variability is a means to handle the inevitable differences among the systems in the family while exploiting the commonalities. In this case, variability enhances the reusability of software. Second, by providing more variability in software systems the flexibility and maintainability of those systems can be improved, as features can be added or adapted – even at runtime – without releasing new products. This can considerably increase the usability of the products.

Meanwhile the increase of variability leads to a situation where the complexity of managing the variability becomes a primary concern during software development that needs to be addressed explicitly by the software development methods and tools. Whereas sound methodic support to analyze and specify variability on the abstract level – e.g. the feature level – is already available, the corresponding support on realization level is still lacking [10]. This holds for the method as well as the tool support.

The realization and management of variability is for some reasons a non-trivial task. A first fact that hampers the consistent management of variabilities is that they often cannot be localized well but have *widespread impacts* down in the implementation documents. This is especially true, if the variability represents a varying quality of the system, as its overall performance, resource demands or interoperability, for instance. As with invariable solutions, a variability has to be addressed on the different levels of abstraction, e.g. architecture, components, subcomponents, classes, etc. to cope with complexity. In addition to this vertical impact, a variability often shows a horizontal impact, i.e. the variability affects several locations spread over the work products on the same level of abstraction. If the interface of a component is affected by a variability, for instance, then the calling components will be affected by the variability in some way too. However, a widespread impact of a variability results in interdependencies among the solution fragments[2] that have to be considered and managed. Furthermore, variabilities may interfere with each other, i.e. the variants[3] offered by the variabilities may exclude or require each other, resulting in further interdependencies. No matter how, the interdependencies caused by variabilities strongly aggravate the consistent and efficient management of the variabilities, as they raise the complexity of the overall solution and have to be considered throughout the whole lifecycle of the variabilities.

Another fact that complicates the management of variability is that variability appears in *manifold forms and realizations*. Generally, a variability extends the problem and consequently the solution space covered by the comprising system. A system that provides variabilities is planned to be applicable in a broader range of problems than its invariable counterparts. Those extensions are

[1] the capability to be changed or adapted

[2] the so-called variation points

[3] potential incarnation of the variability

neither restricted to certain problems nor to special solutions. In principle, every solution in a software system can be kept variable. A whole string of techniques and mechanisms to realize variability [13][11][17] in the various solution documents are already available, especially to handle variability on the code level but also on the upper levels of abstraction, the architecture for instance. Unfortunately, the impacts of the different realizations are not completely understood yet and there is consequently only little methodic support in the realization and management of variability.

This paper concentrates on the more product-family-related issues of variability management. The experiences we have made with variability management in various domains (building automation, embedded operating system, automotive), give us reason to believe, that the management of variability can be facilitated substantially, if we find a general model of how variability is realized and handled in product families that holds for all kind of variability throughout all abstraction levels. Such a model should:

- provide well-defined concepts to foster a common understanding of variability and its impacts
- identify common issues in the handling of variability, e.g. traceability, variable binding times and evolution
- and thus ease the development of variability aware software development methods and tools

Unfortunately, such a model is still missing, although the required terminology has already been defined quite well [19]. As a consequence, different approaches and slightly differing notions are used to realize and handle variability on the diverse abstraction levels, e.g. architecture, source code, and documentation, which inhibits synergistic effects to appear and complicates the consistent management of variability considerably.

In order to approach such a model, this paper discusses the interrelationships between the specification and realization of variability, identifies appropriate concepts and interrelates them in form of a general model of variability in product families. In addition to this model, the paper outlines an application of the model: our language to specify variability in product family assets.

The remainder of the paper is structured as follows: Section 2 discusses variability in product families. Besides the different motivations for variability, the two levels on which variability is approached are described. Section 3 illustrates the various incarnations of variability in the product family assets and identifies common properties among them. These commonalities in the realization of variability led to our model of variability in product families that is presented in section 4. Section 5 outlines an instantiation of the model: the Variability Specification Language. The paper closes with a conclusion.

2. Variability in Product Families

Product family[4] engineering [14] is a commonly accepted approach to exploit the reuse potential of similar software systems in a systematic and pre-planned way. The rationale behind this approach is to identify common solutions parts in a set of envisioned systems, which only have to be implemented once as so-called assets[5] and can be reused afterwards during the construction of the manifold family members in application engineering processes. This leads to the characteristic development process (six-pack) with the two development tracks: domain engineering (development for reuse) and application engineering (development with reuse).

Commonly, a product family comprises a reference architecture and a string of components. In addition to design and implementation documents, other kinds of assets as requirement specifications, test processes and data, production plans or domain knowledge can be supplied through the family as well depending on their reuse potential. The overall success of a product family approach, however, is closely coupled with the capability to handle the required differences among the family members in a consistent but also economic way. To this end, the family and its members are designed to be variable, i.e. they provide variabilities.

Generally speaking, a variability represents a capability to change or adapt system [19], i.e. the system facilitates certain kinds of modifications. Such a change or adaptation can affect the behavior of the system as well as its qualities. From a more technical perspective of a software engineer, a variability is a means to delay a (design) decision to a later phase in the lifecycle of the software system [19]. If a decision among a set of possible variants cannot be taken at a certain time during the development of the system, then a generic solution has to be realized in the work products at hand that allows to take the decision later on.

An analysis of the driving forces behind variability in software systems in general and product families in special reveals that two *main motivations* can be distinguished:

- **Usability**. By providing variability in a software system, the flexibility and maintainability of the system can be improved, as features can be added or adapted – even at runtime – without releasing new products. This can increase the usability of the products considerably.

[4] group of systems built from a common set of assets[4] [4]

[5] partial solution, such as a component, a design document or knowledge that engineers use to build or modify software products [21]

- **Reusability**. Variability has been recognized as the key to systematic and successful reuse. Especially in family-based approaches like software product families, variability is a means to handle the inevitable differences among systems in the family while exploiting the commonalities and thus increases the reusability of software.

The distinction between both motivations is necessary – although often neglected –, because the respective variabilities are handled differently and influence the software development processes in different ways. In case of increased usability, which can be generally of interest in any software development approach, the respective variability is used to handle an intra-product variation [11] and thus is a feature of the product, i.e. the product contains a mechanism to handle the variability dynamically after the delivery of the product to the customer. Apparently, such dynamic variabilities in principle require no special treatment during the development of the software systems as the can be realized and handled like any other feature of the system. The main issues raised by dynamic variabilities are the mastering of the increased functional complexity and the available implementation mechanisms. The increased reusability, on the other hand, can be considered as a peculiarity of family-based approaches. In this case, variability is used to handle the differences between the members of a family (inter-application variability). Obviously, such a variability is not a feature of the family members but of the comprising family and is handled statically, i.e. once bound to a distinct variant during the derivation of a family member, the variability vanishes and is no longer existing in the family member. Static variabilities affect the development processes considerably and raise a string of new issues, e.g. configuration and instantiation support, management of variants, evolution support etc.

It has to be pointed out, that the above-mentioned motivations do not exclude each other, but can coincide in one variability. In this case, the respective variability will support several binding times[6], and the handling of the variability will therefore depend on the actual binding time of the variability in the application engineering processes. If the corresponding decision is taken early enough in the software development process, then the variability is handled statically, i.e. the work products will be tailored according to the decision, otherwise it will be handled dynamically. A variable binding time allows to handle the trade-off between tailored, highly efficient solutions on the one-hand and flexible but more complex ones on the other. To subsume, from a product family perspective we have to face two motivations of variability: increased usability and reusability, whereas the latter considerably affects the development methods and tools and leads to peculiar issues. The increased usability is primary of interest if it coincides with attempts to increase the reusability of the work products. Consequently, the remainder focus of this paper focuses on static variabilities.

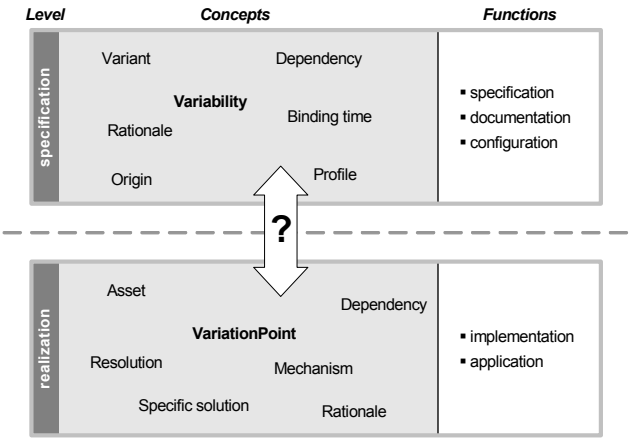

Figure 1. Two levels of variability handling

In family-based engineering approaches, variability is typically approached on two different levels of abstraction (cf. fig. 1): on the specification and the realization level. A distinction between those both levels is sensible, since they fulfill different functions and use different concepts to represent variability.

On the *specification level*, the involved stakeholders put their focus on the externally visible characteristics of variability and suppress realization details. The requirements and knowledge about the variabilities in the family are captured and represented by means of feature models [15] or dedicated variability models [7][20]. These models comprise information about the variabilities themselves, e.g. their origins, the range of offered variants, the reuse potential of the variants and furthermore information about the interdependencies among the variabilities, and information concerning the binding of the variability, e.g. the supported binding times and the roles that can bind a variability. In most cases, concepts of the problem space are used to express information about variability. The main modeling concepts used to represent variabilities are variable features (in the feature models) or variabilities themselves. Besides the information about the supported variabilities, there will also be information about the family members that are instantiated in the product family. This information is captured in application models or profiles that keep track of the variability-related decisions, which were taken during the configuration of the family members and control the resolution of the static variabilities in the application engineering. The information about variability on the specification level is used for various purposes. First, it is a means to analyze and specify the requirements for the implementations. Second, it documents the capabilities offered by the family on an abstract level, and thus is the entry point to understand the family

[6] phase in the development process in which the variability is bound to a certain variant

and its members. Third, it forms the basis for the configuration and instantiation of family members [12].

On the *implementation level*, i.e. in the set of reusable assets provided through the product family[7], the software engineers have to realize and handle the required variability that has been specified on the specification level. To this end, they identify the impact of the variabilities in the various software assets offered through the product family and support the demanded variation by using appropriate mechanisms. In the application engineering processes, the application engineers deploy the static variabilities to derive specific solutions. During this derivation, the static variabilities are resolved to specific solutions. The main concept that represents variability on the implementation level is the variation point. A *variation point* is a spot in a software asset where variation will occur [13][19], i.e. where a variability is realized, at least partially. Thus, a variation point can be considered as some kind of generic element in a software asset. This is especially true, if the variability is motivated by reuse concerns.

Whereas sound methodic support to analyze and specify variability on the specification level is already available, the situation on the implementation level is quite different. Although a whole string of variability mechanisms exits to realize variability in the variation points (at least in the source code assets), e.g. appropriate language constructs, pre-processors, external generators etc., only few methodological and tool support is available that meets the rising demands of variability management. Thus, the mapping between the two levels (illustrated through the question mark in fig. 1.) and the management of variability on the realization level often remains a highly creative, individual and consequently complicated task. In order to cope with the rising complexity induced through variability, more systematic approaches are required. To this end, a general model of variability in product families is required, which identifies concepts, issues and patterns that can be applied throughout the whole lifecycle of a product family. Before we present our model, we first take a closer look at the implementation level of variability to reveal commonalities in a way variability is realized in the various asset types.

3. Variability on the Implementation Level

Within a product family any kind of work product used to construct a software system can be provided as a reusable software asset. Generally, some of them are not affected by variability – i.e. they are used as is in every member of the family –, but they usually form the minor part. Most of the assets are influenced by variability in one or the other way (illustrated through the grey triangle in fig. 2). Since the impact of a variability is neither limited to certain abstraction levels nor to distinct asset types, any asset provided through a product family can in principle contain variation points. Examples for such software assets are generic requirement templates, reference architectures, components, source code, test cases and even generic documentation assets (cf. fig. 2).

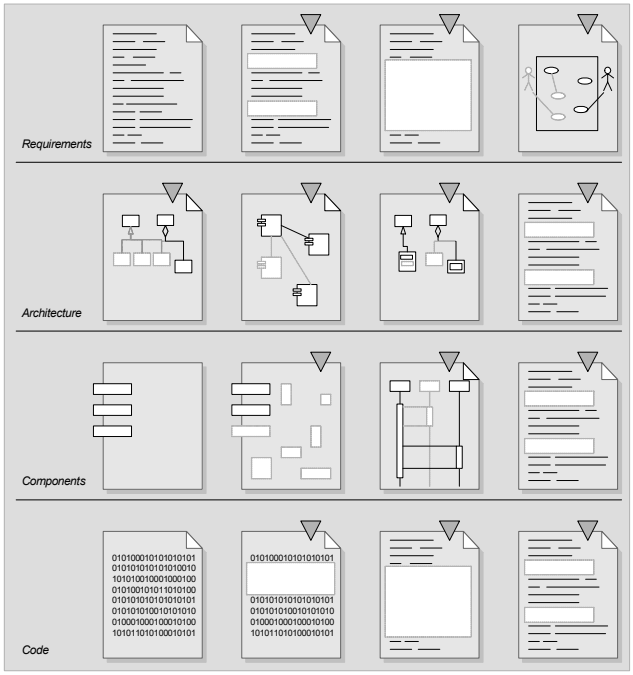

Figure 2. Various asset types in a product family

Apparently, there are different ways to represent the information contained in the assets. The information can be expressed through text, diagrams and binary data and each of these representations can contain variation points (cf. fig. 2). In recent years, especially variation points in diagrams attracted the attention of industry [18][16] and academia [9][2], as variability had to be implemented on the architectural level too, in order to allow for reuse in the large. Regarding the granularity of a variation point it can be stated, that a variation point can extend from multiple files, e.g. in case of software components, over document fragments like blocks, lines or diagram elements down to single information items, as characters or bytes. To summarize, variation points can appear in manifold ways in software assets, which complicates the management of the variabilities considerably, especially if they show widespread impacts.

Although the various incarnations of variation points differ substantially (cf. fig. 2), they also share some common properties. If we abstract from the different asset contents and the concrete realizations of variation points we observe the following *common functions* of variation points:

[7] the implementation level of variability (all assets affected by variability on the different levels of abstraction) should not be confused with the implementation level of the product family (only code assets).

- **Localisation**. A variation point localizes a variation in an asset.
- **Abstraction**. From an external point of view, i.e. by suppressing internal realization details, a variation point abstracts from the specific realizations of the variants.
- **Specialization**. In addition to the abstraction, a variation point supports its specialization to a concrete solution in an appropriate way. To achieve this, it provides a specification that describes how to specialize the variation point to a distinct variant and a mechanism that realizes the specialization. In order enable variation, the specification of the specialization must be parameterized by the variabilities in some way, i.e. the specification must be a function of the variabilities.

Besides the aforementioned common functions, also *desirable features* can be identified that any variation point should have in order to render its functions and retain manageable (cf. [1]):

- **Identification**. It should be evident what part of the asset is immutable and what part is affected by variabilities. That way, the added complexity has only a limited impact in the asset.
- **Clear Structure**. Variation points in the assets should be structured as clearly as possible. First, they should not obscure the structure of the comprising asset. Second, if necessary, variation points should be structured in a hierarchical way, i.e. they should not overlap partially.
- **Expressiveness**. Along with the variation point its specialization must be specifiable. This is of special interest in the case of variation points that implement static variability, where the specialization is often carried out manually.
- **Localized**. The impact of a variability should be as localized as possible, i.e. the variation points should be designed and implemented in a way that concentrates the impact of the variability to as few points as possible.
- **Tracability**. Bidirectional traces between variabilities and the variation points that implement them must be maintainable in order to interrelate the two abstraction levels. Additionally, traces between the variation points that implement the same variability must be maintainable as well, in order to allow the consistent evolution of a variability.

In spite of the considerable differences between the various realizations of variability, e.g. in the way a variation point localizes variability and the way it supports its specialization in detail, apparently the commonalities among the variation points are substantial. The realization of this led to our model of variability, which is presented in the next section.

4. A Model of Variability in Product Families

In order to pave the way towards more systematic and consequently more efficient approaches to manage variability, we have developed a general (meta-)model of variability in product families that identifies and interrelates the concepts on the two abstraction levels mentioned in section 2. The motivation behind this model was:

- to provide concepts to foster a common understanding of variability and its impacts,
- to identify common issues and patterns in the handling of variability, and finally
- to ease the development of variability aware methods and tools

In fig. 3. you find an excerpt[8] of our model, which will be explained in the following.

The upper box at the right side addresses variability on the *specification level*. The main concepts are Variability and Profile. A *Variability* represents a variability in the ProductFamily and provides a Rationale and a Range of Variants. Between the Variants Dependencies, e.g. requires or excludes relationships, can be stated. As the Variants are associated with Variabilities, the Dependencies consequently concern the respective Variabilities. Furthermore, a Variability provides information about its supported BindingTimes.

A *Profile* keeps track of the variability-related decisions that were taken during the configuration of a family member. Thus, it specifies or identifies a member of the family. A Profile comprises a set of Assignments that can be accessed via the Variability. Each assignment represents a taken decision, e.g. Variant A has been chosen for Variability B at the BindingTime C. If no Assignment is available for a Variability, then the Variability is unbound in the profile.

The lower box at the right side addresses variability on the *realization level*. The main concept is the *VariationPoint*. The Assets provided through the ProductFamily can contain VariationPoints. A VariationPoint implements a Variability of the specification level, at least partially. Usually, a Variability causes several VariationPoints that are spread over multiple Assets. The concrete number of VariationPoints caused by a Variability depends of course on the Variability itself and the Assets provided through the ProductFamily. On the other side, a VariationPoint can be affected by more than one Variability. In this case, the impacts of the Variabilities overlap. Consequently, the multiplicity of the relationship between Variabilities and VariationPoints is n:m.

Local dependencies, i.e. Dependencies between the VariationPoints that are not already expressed through the Dependencies on the specification level, can be stated on the realization level. However, in order to keep the num-

[8] the complete model will be presented in our PhD thesis

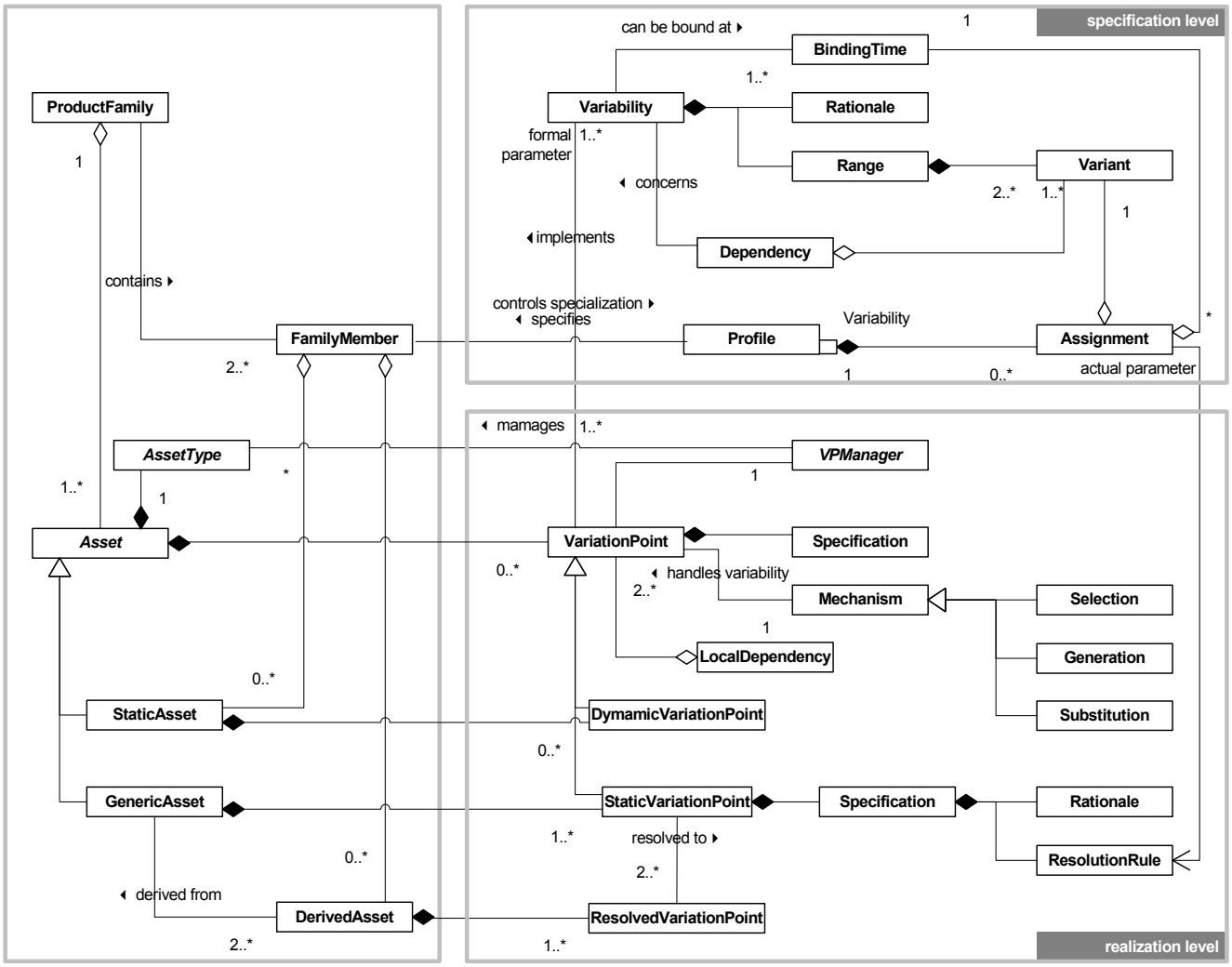

Figure 3. A general model of variability in a product family

ber of dependencies and the effort to manage them as small as possible, dependencies should be specified globally on the specification level, if possible. Dependencies that result from the fact, that VariationPoints realize the same Variability, do not have to be expressed explicitly, they can be derived from the association between Variability and VariationPoint.

A VariationPoint is associated with a Mechanism that handles the Variability. Various Mechanisms can be used to this end. The Mechanisms can be coarsely[9] categorized into three classes [5][6]: Selection, Generation and Substitution. By means of a Selection mechanism, an existing solution can be selected to specialize the variation point. The corresponding specification of the specialization is illustrated in fig. 4. Exemplary selection mechanisms are if/else or switch constructs in preprocessor and programming languages, or inheritance in object oriented languages. A generative mechanism allows the generation of a solution, e.g. through an external generator. The specialization specification forms the input of the generator and the generated output specializes the variation point. Substitution mechanisms are rather simple; they support the specialization of the VariationPoints by unique, externally provided solutions. Therefore, the corresponding variation points can be considered as some kind of gap.

As stated in section 2, two different motivations can be identified for a Variability. Those motivations lead to different types of VariationPoints. The first one, the *DynamicVariationPoint* demarcates a solution in an Asset that allows to handle the Variability late in the lifecycle of the product, i.e. after the delivery. Consequently, DynamicVariationPoints are not specialized during the design of the corresponding FamilyMember. In contrast to them, a *StaticVariationPoint* has to be specialized during the design and implementation of the FamilyMember. The result of such a resolution is a ResolvedVariationPoint, which no longer supports variation. In order to support

[9] more detailed taxonomy of such mechanisms can be found in [17]

their specialization, StaticVariationPoints provide a Specification, which contains a Rationale and a ResolutionRule. The specialization can be automated through an appropriate mechanism. To facilitate the evolution of a variability realization, the association between StaticVariationPoint and ResolvedVariationPoint should be maintained in the ProductFamily, in order to propagate changes in both directions.

StaticAssets contain no StaticVariationPoints. Thus, they can be used in the application engineering without any specialization. GenericAssets on the other hand contain at least one StaticVariationPoint. The specialization of a GenericAsset results in a DerivedAsset that is used to construct the FamilyMember. DerivedAssets contain no StaticVariationPoints but only ResolvedVariationPoints.

Variabilities control as formal parameters the specialization of the VariationPoints. What serves as actual parameters depends on the type of the VariationPoint. In the case of a DynamicVariationPoint, the specialization is controlled by runtime parameters in the software system. With StaticVariationPoints the assignments in the profiles form the actual parameters of the specialization. If the ProductFamily supports several BindingTimes for a Variability, then the specialization specification of the resulting variation points may also depend on the variability's binding time, e.g. the conditions in a selection (cf. exemplary condition 3 in fig. 4. above). Hence, the variation point's specialization specification is not only a function of the corresponding variabilities but also of their actual binding times.

As illustrated in the model, the only two associations between concepts on both levels are the implements association between Variability and VariationPoint and the association between the Assignments and the ResolutionRules. The first association is established during the implementation of the assets and has to be maintained during the whole lifecycle of the ProductFamily. Along this association, information can be propagated between the both abstraction levels. The second association does not need to be maintained explicitly. It can be derived from the first one. If the actual parameters have to be determined for the specialization of a StaticVariationPoint, then the corresponding assignments can be retrieved from the profile through the variabilities associated with the VariationPoint. Obviously, the first association is of utmost importance for any product family approach. Bidirectional traces between the variabilities and the variation points must be expressible and maintainable in an efficient way. As a prerequisite, the variation points – static as well as dynamic ones – must be identifiable in the assets.

To support the management of variability on the implementation level, VPManager instances can and should be provided for the different AssetTypes of a ProductFamily. A VPManager is a tool that supports the domain and application engineers in the various variability-related tasks, as implementation, identification, resolution, assessment, and evolution of variation points in assets of the respective types. The VPManager class in the model captures the management-related issues and solution patterns or principles, e.g. the resolution in case of variable binding times or the automated evolution of a variabilty. A lot of methodical and tool support is conceivable and required to this end, but only few is available yet.

Specification of a selection:

```
if (condition1) solution1
elif (condition2) solution2
…
elif (conditionN) solutionN
else default-solution
```

Exemplary conditions:

```
1. VariantA
2. VariabilityA.VariantB and
   not VariabilityB.VariantD
3. VariabilityA.BindingTime < BindingTime.IntDes
```

Figure 4. Specification of a selection

5. Instantiation of the Model: Variability Specification Language

Based on the above-mentioned meta-model and the identified demands for variation points, we have developed a language to specify variability in product family assets – the Variability Specification Language (VSL) – and appropriate tools (processor, viewer). VSL is an XML-based language that can be applied in a broad range of documents and thus allows to handle variability in a uniform manner. Besides the previous drivers, VSL has been inspired by the frame technology [3] and the popular C preprocessor. Both of them can be considered as macro languages and the same applies to VSL – at least partially – too.

VSL first of all allows to specify the impacts of variabilities in the assets, i.e. the variation points. Besides the clear identification of the variation points and the variabilities that affect them, the specialization of the variation points can be formulated as well. To this end, VSL provides markup to specify the selection of pre-built variants and the generation (up to now XSLT and JScript are supported) or the substitution of specific solutions and hence supports the basic mechanisms to handle variability.

Based upon the VSL-specifications, specialized solutions (XML or text documents) can be derived from the VSL-based generic assets during the application engineering. This resolution is controlled by profiles, which can be expressed by means of VSL too (cf. fig. 5). Besides the values of the variabilities, VSL specifications can take the variabilities' binding time into consideration. Although the main driving force behind VSL was to support static variability, VSL can be applied with dynamic variability

Profile:

```
<vsl:profile id="StdCfg" vm="prosekko">
  <vsl:set var="Status" bt="ReqSpec">extended</vsl:set>
  <vsl:set var="PreemptiveMultitasking">yes</vsl:set>
  <vsl:set var="ConformanceClass">ECC2</vsl:set>
  …
  <vsl:set var="Tasks" bt="IntDes">
    <task>…</task> …
  </vsl:set>
  <vsl:set var="Resources" bt="IntDes">3</vsl:set>
</vsl:profile>
```

Asset:

```
<vsl:import href="../include/debug.h.vsl" once="yes"/>
…
<vsl:select var="Status">
  <vsl:option value="basic"/>
  <vsl:option value="extended">
  int resource_occupied[<vsl:subst var=""/>]
     [<vsl:subst var="Resources"/>];
  </vsl:option>
</vsl:select>
```

Figure 5. A VSL document and profile fragment

as well. In this case, the VSL markup is not processed by the VSL-processor, but merely serves for identification and specification purposes. A more detailed discussion of the VSL features can be found in [8].

The main advantages in applying VSL to specify variability in a product family can be seen in the uniform and explicit treatment of variability. First, the language can be used to specify the variability in the different asset types. This considerably eases the development of special variability management tools, e.g. to facilitate the evolution of variability, that can be applied throughout the whole product family engineering process. Second, due to the explicit specification of the variability by means of a dedicated language it gets quite easy to identify and assess the impacts of a variability down in the assets. A general advantage of VSL – as with all XML-based approaches – is the extensibility of the language and the remarkable tool support. Although still being in a evolving state, VSL has already proven the feasibility of XML-based variability management. It has been deployed successfully to handle the variability in an embedded operating system on the requirements and the code level (C-Code). In an industrial context we have deployed VSL to specify variability on the architecture level in UML-diagrams.

6. Conclusion

The increased amount of variability in software systems meanwhile requires more systematic approaches to cope with the rising complexity introduced through variability. This is especially true in product families, where variability is a means to handle the inevitable differences among the systems in the family while exploiting the commonalities. Widespread impacts of variability and the various realizations considerably complicate the management of variability in product families. In order foster more systematic and consequently more efficient approaches of variability management we have discussed the commonalities and differences of variability in product families, identified appropriate concepts and interrelated them in form of a general model of variability in product families. The model has been applied to develop a small language to specify and realize variability in product family assets.

We believe that the management of especially static variabilities, which can be considered as a main characteristic of product family approaches, is an issue that can and should be addressed in an explicit and overall manner to keep track with the rising complexity. To achieve this, a common understanding and management of variability is required across the various asset types. The presented approaches intent to pave the way towards this.

References

[1] Bandinelli, S.: Product Family Engineering with XML, Proc. of Dagstuhl Seminar No. 01161 Product Family Development, Wadern, Germany, 2001

[2] Bachmann, F.; Bass, L.: Managing Variabilities in Software Architectures, Proc. of 2001 Symposium on Software Reusability, Toronto, Ontario, Canada, May 2001

[3] Bassett, P.G.: Framing Software Reuse - Lessons From the Real World, Yourdon Press Computing Series, 1997

[4] Bass, L.; Clements, P.; Donohoe, P.; McGregor, J.; Northrop, L.: Fourth Product Line Practice Workshop Report, http://www.sei.cmu.edu/publications/documents/00.reports/00tr002.html, November 1999

[5] Baum, L.; Becker, M.; Geyer, L.; Molter, G.: Mapping Requirements to Reusable Components using Design Spaces, Proc. of IEEE Int'l Conference on Requirements Engineering (ICRE 2000), Chicago, USA, 2000

[6] Becker, M.: Generic Components: a symbiosis of paradigms, 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00), 2000

[7] Becker, M.; Geyer, L.; Gilbert, A.; Becker, K.: Comprehensive Variability Modelling to Facilitate Efficient Variability Treatment, Fourth International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, October 2001

[8] Becker, M.: XML-Enhanced Product Family Engineering, Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology (IDPT2002), Pasadena, USA, June 2002

[9] Bosch, J.: Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach, Addison-Wesley, 2000

[10] Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, H.; Pohl, K.: Variability Issues in Software Product Lines, Proc. 4th Int'l Workshop on Product Family Engineering, Bilbao, Spain, 2001

[11] Czarnecki, K; Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications, Addison-Wesley, 2000

[12] Geyer, L.; Becker, M.: On the Influence of Variabilities on the Application Engineering Process of a Product Family, Proceedings of the 2nd the Second Software Product Line Conference, San Diego, USA, 2002

[13] Jacobson, I.; Griss, M.; Jonsson P.: Software Reuse - Architecture, Process and Organisation for Business Success, ACM Press / Addison-Wesley, 1997

[14] Jazayeri, M.; Ran. A; Van der Linden, F.: Software Architecture For Product Families: Putting Research into Practice, Addison-Wesley, May 2000

[15] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990

[16] Muthig, D.; Atkinson, C.: Model-Driven Product Line Architectures, Proc. of the Second Software Product Line Conference, LNCS 2379, Springer, San Diego, USA, August 2002

[17] Svahnberg, M.; Van Gurp, J.; Bosch, J.: A Taxonomy of Variability Realization Techniques, Technical paper, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002

[18] Thiel, S.; Hein, A.: Systematic Integration of Variability into Product Line Architecture Design, Proceedings of the Second Software Product Line Conference, LNCS 2379, Springer, August 2002

[19] Van Gurp, J.; Bosch, J.; Svahnberg, M.: On the Notion of Variability in Software Product Lines, Proceedings of WICSA 2001, August 2001

[20] Voget, S.; Angilletta, I.; Herbst, I.; Lutz, P.: Behandlung von Variabilitäten in Produktlinien mit Schwerpunkt Architektur, Proceedings of 1. Deutscher Software-Produktlinien Workshop (DSPL-1),, Kaiserslautern, Germany, November 2000

[21] Withey, J.: Investment Analysis of Software Assets for Product Lines, http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.010.html, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1996