

Multi-Dimensional Variability Modeling

Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, Gunter Saake
School of Computer Science
University of Magdeburg, Germany
{rosenmue,nsiegmun,tthuem,saake}@ovgu.de

ABSTRACT

The variability of a *software product line (SPL)* is often described with a feature model. To avoid highly complex models, stakeholders usually try to separate different variability dimensions, such as domain variability and implementation variability. This results in distinct variability models, which are easier to handle than one large model. On the other hand, it is sometimes required to analyze the variability dimensions of an SPL in combination using a single model only. To combine separate modeling and integrated analysis of variability, we present VELVET, a language for multi-dimensional variability modeling. VELVET allows stakeholders to model each variability dimension of an SPL separately and to compose the separated dimensions on demand. This improves reuse of feature models and supports independent modeling variability dimensions. Furthermore, VELVET integrates feature modeling and configuration in a single language. The combination of both concepts creates further reuse opportunities and allows stakeholders to independently configure variability dimensions.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications

General Terms

Design, Languages

Keywords

Feature models, variability modeling, separation of concerns

1. INTRODUCTION

A *software product line (SPL)* is a set of similar software products that can be distinguished in terms of *features* [11]. Features are used to describe commonalities and variability of an SPL. A *feature model* is a hierarchical representation of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VaMoS '11 January 27-29, 2011, Namur, Belgium

Copyright 2011 ACM 978-1-4503-0570-9/01/11 ...\$10.00.

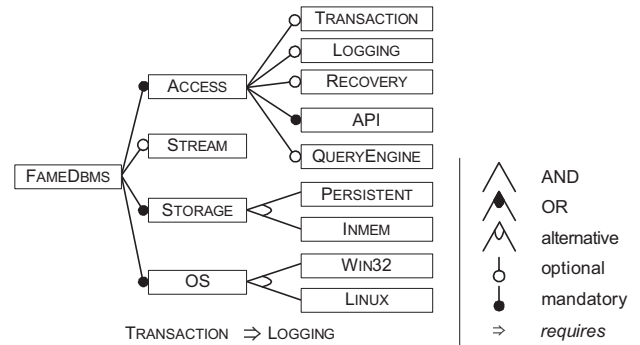


Figure 1: Excerpt of the FAMEDBMS feature model.

the features of an SPL [18]. In Figure 1, we show an excerpt of the feature model of FAMEDBMS¹, an SPL of *database management systems (DBMS)* [30]. A feature model has a tree structure with parent-child relationships between features. For example, feature STORAGE in Figure 1 represents the functionality for data storage. It has two alternative child features that represent different storage mechanisms. Propositional formulas describe additional dependencies between features that cannot be encoded in the feature hierarchy. For example, feature TRANSACTION in Figure 1 *requires* feature LOGGING. Such *domain constraints* restrict the variability to avoid invalid configurations with respect to the application domain.

A feature model thus captures the variability of an SPL's application domain. However, an SPL also contains domain independent variability such as *external features* [37]. Examples are the execution environment, security requirements, and the context at runtime (e.g., the location of a mobile device). Moreover, there can be *internal variability*, such as implementation variability offered by components [24]. A product of an SPL has to satisfy requirements along these additional *variability dimensions* as well.

In current research, the variability dimensions of an SPL are sometimes separated from each other and are sometimes mixed in a single feature model. For example, the operating system features in Figure 1 (subfeatures of OS), do actually not describe variability of the application domain but represent external features of the execution environment. On the other hand, some variability dimensions are commonly separated from each other. For example, implementation-specific variability is usually not shown in a feature model.

Integrated as well as separate modeling of variability di-

¹<http://fame-dbms.org>

mensions have benefits and drawbacks. A single integrated feature model can simplify communication between stakeholders and analysis of variability, but it hinders communication when the model gets too complex [27, 2]. Using a separate feature model for each variability dimension enables stakeholders to handle complex feature models [35] and to reuse individual variability models in different SPLs. However, separate models complicate analysis and communication of the entire variability of an SPL [26, 32].

To overcome these problems, we propose to model the variability dimensions of an SPL separately and to compose the dimensions on demand. Furthermore, an SPL configuration can be expressed as a set of constraints that restrict the variability [10]. Hence, the same mechanism, namely constraints over the features of an SPL, can be used for both, feature modeling and SPL configuration. We thus consider SPL configuration as a variability dimension along which variability is reduced at configuration time.

With VELVET, we present a language that seamlessly integrates feature modeling and SPL configuration. We make two contributions. First, we enable SPL engineers to define separate feature models for the variability dimensions of an SPL. We provide different mechanisms for composition of feature models, each having different benefits. Second, we describe configuration steps with *configuration constraints* [28] to integrate feature modeling and configuration in a single language. The combination of multi-dimensional modeling and configuration allows us to improve reuse of feature models, to manage complex models, and to independently model and configure variability dimensions of an SPL.

2. MULTI-DIMENSIONAL VARIABILITY MODELING

Multi-dimensional separation of concerns (MDSOC) [34] is a concept for separating multiple important concerns in software development. In this paper, we apply the MDSOC concept to variability modeling. Our goal is to provide a way to model different *variability dimensions* separately and to integrate variability modeling with SPL configuration.

2.1 Variability Dimensions

A variability dimension is a kind of variability that is important for a stakeholder. Examples are the application domain, the execution environment of a program (e.g., the operating system, the hardware), the context at runtime (e.g., time, space, the user, etc.), non-functional properties (e.g., security, quality of service), and implementation variability. An SPL’s feature model can be decomposed into arbitrary dimensions. For example, we can decompose the application domain of a DBMS into a dimension for query processing, one for storage management, and one for transaction management. Separate modeling of an SPL’s variability dimensions can have several benefits:

- **Reuse:** Some variability dimensions of an SPL are application or even domain independent. Decomposition of a feature model thus enables reuse of individual dimensions across different SPLs.
- **Managing complexity:** Splitting a large complex model into manageable pieces eases distributed modeling and navigation and enables separate analysis of variability dimensions.

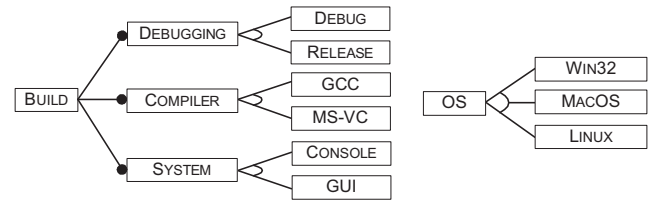


Figure 2: Platform-specific variability dimensions Build and OS.

- **Independent modeling:** Separation of variability dimensions allows experts to model them independently. For example, a security expert can focus on the security dimension only.
- **Independent configuration:** Based on a decomposed variability model, different users may configure variability dimensions independently.
- **Crosscutting variability:** In the context of large systems built from multiple interdependent SPLs (a.k.a. *Multi Product Lines* [28]), we may want modularize and configure variability that crosscuts several SPLs (e.g., security aspects).

We depict two examples for reusable domain-independent variability dimensions in Figure 2. The models describe variability that is specific to the build system (BUILD) and the operating system (OS). Modeling all variability dimensions of a DBMS in a single feature model results in one large model. Instead, we can split the feature model into a domain model with DBMS specific features (DBMS features in Figure 1), a model for the operating system (OS in Figure 2), and a model for the build system (BUILD in Figure 2).

In practice, two or more variability dimensions may overlap. That is, some features are part of multiple dimensions. For example, the implementation features of an SPL may be a superset of the SPL’s domain features. Furthermore, there can be optional and alternative dimensions. For example, if we have two different implementations of the same SPL (e.g., using different programming languages), we may also have two different implementation variability models. We derive the complete variability model of an SPL by *superimposing* all variability dimensions or a subset of the available dimensions.

Generalization and Specialization. The modeling concepts that we describe in this paper, require to modify existing variability models (e.g., to add features to an existing model). We distinguish between the two basic modification mechanisms *generalizations* and *specializations* [36]:

Generalization: Generalization means to increase variability by adding new features to a model. For example, adding an alternative feature increases the variability and thus generalizes the SPL.

Specialization: Specialization means to reduce variability by adding constraints that encode dependencies between features. For example, implementation dependencies (e.g., mutually exclusive features) reduce variability and thus specialize an SPL.

Usually, we use a combination of generalization and specialization to model extension, composition, and configuration of variability dimensions.

2.2 Modeling SPL Configuration

The SPL configuration process can be represented as a modification of the feature model that reduces the variability. It can occur in a stepwise manner, which is known as *staged configuration* [12]. Each configuration step results in a more *specialized SPL* that represents less products and finally only a single product. A configuration step is a *specialization* according to our definition above. It can be described by adding constraints to a feature model [10]. For example, a simple feature selection during configuration means to add a *configuration constraint* that *requires* the selected feature.

In general, a configuration constraint is a propositional formula over the set of features. It is not different than a usual variability constraint (e.g., a domain constraint) and avoids invalid configurations with respect to a concrete application scenario. We thus describe the configuration process as a special variability dimension that does not add new features. A configuration step may reduce the variability along multiple dimensions or along a single dimension only. For example, a security expert may configure only the features of a security dimension. The separation of variability dimensions thus helps to simplify the configuration process and enables configuration of selected variability dimensions only. By separately defining and composing configuration steps, we provide a structured mechanism to reuse configuration decisions.

3. VARIABILITY MODELING WITH VELVET

There are several languages for textual variability modeling [1, 5, 6, 9, 15]. In the following, we present VELVET, a language that generalizes variability modeling concepts: It allows domain engineers (1) to describe variability dimensions in separate models, (2) to compose variability models, and (3) to describe partial configurations (i.e., specializations) using the same concepts. The syntax of VELVET uses parts of the syntax of TVL (text-based variability language) [9]. For example, we use keywords `oneOf` and `someOf` to define alternative features and or-groups. However, we introduce concepts based on object-oriented programming (OOP) and argue that it is reasonable to use a syntax similar to other OOP languages. We thus use parts of the C# syntax where applicable.

We depict an excerpt of the grammar of VELVET in Figure 3. The complete grammar and a parser can be found on the MultiPLe website.² In the following, we focus on feature models without attributes. However, the complete grammar also includes feature attributes such as strings and numerical values.

3.1 Modeling Variability Dimensions

With VELVET, we can describe the variability dimensions of an SPL in separate variability models. In Figure 4, we show an example for the FAMEDBMS model of Figure 1. The definition of a variability model with VELVET is similar to an OOP class definition. It consists of a declaration of the described concept (i.e., the root of the feature model) using the keyword `concept` (line 1 in Figure 4). Within a concept, we use a hierarchical definition of the features according to the feature model (lines 2–16). Each feature is

²<http://fosd.de/multiple/>

```

concept.g
1 concept: ["refines"] "concept" cName [cBase] cBody
2 cName: ID
3 cBase: ":" cName ("," cName)*
4 cBody: "{" definition* "}"
5 definition: (feature|f-group|constraint|instance)

feature.g
6 feature: ["mandatory"] "feature" fName feature-def
7 fName: ID ( "." ID)*
8 fGroup: or-group | alt-group
9 or-group: "someOf" group-childs
10 alt-group: "oneOf" group-childs
11 group-childs: "{" feature+ "}"
12 feature-def: "{" definition* "}" | ";"

constraint.g
13 constraint: "constraint" [cons-name=""] cons-def ";"
14 cons-name: ID
15 cons-def: fName | "!" cons-def | "(" cons-def ")"
16 | cons-def cons-op cons-def;
17 cons-op: "&&" | "||" | "->" | "<->";

instance.g
18 instance: cName instName ";"
19 instName: ID

```

Figure 3: Excerpt of the grammar of VELVET.

defined with keyword `feature` and a name. There can be multiple features with the same name as long as they are distinguishable via their fully qualified name, which is the path in the feature tree. In contrast to TVL, features are by default optional and keyword `mandatory` denotes mandatory features (line 3). This is required to achieve consistency of language mechanisms that are based on separating generalization (adding features) and specialization (adding constraints). By defining a feature as optional, we do not apply any constraints, which corresponds to unrestricted generalization. Adding keyword `mandatory` corresponds to adding a constraint that restricts the variability (i.e., it is *required* to select the mandatory feature when its parent is selected). Other keywords that apply constraints are `oneOf` (line 12) and `someOf`, which denote groups of features with alternative and inclusive-or relation respectively.

As shown in line 8 of Figure 4, we define constraints between features using the keyword `constraint`, an optional name (`txn`), and a propositional formula. We support the boolean operators negation (`!`), conjunction (`&&`), disjunction (`||`), implication (`->`), and biconditional (`<->`). To simplify constraint definitions, a constraint can be nested within a feature definition. This means that the constraint has to be true only when the enclosing feature is selected, which can be transformed in a usual constraint using an implication. For example, constraint `txn` in line 8 only has to be true when feature `ACCESS` is selected.

3.2 Variability Model Composition

We provide three alternative mechanisms to compose variability models. Each mechanisms has benefits and drawbacks and their combination is needed for multi-dimensional variability modeling, as we discuss in Section 4.

- **Inheritance:** As in OOP, we use inheritance to create a new variability model that extends an existing model with new features and constraints. We support multiple inheritance to compose existing variability models.
- **Superimposition:** We adopt the concept of superim-

```

1 concept FameDbms {
2   mandatory feature Access {
3     mandatory feature API;
4     feature Transaction;
5     feature Logging;
6     feature Recovery;
7     feature QueryEngine;
8     constraint txn = Transaction -> Logging;
9   }
10  feature Stream;
11  mandatory feature Storage {
12    oneOf { feature Persistent; feature InMem; }
13  }
14  mandatory feature OS {
15    oneOf { feature Win32; feature Linux; }
16  }
17
18  constraint win = OS.Win32 -> !Access.QueryEngine;
19 }

```

Figure 4: The variability model of FAMEDBMS in VELVET.

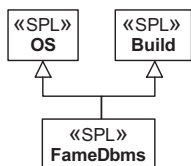


Figure 5: Composition of variability models with multiple inheritance.

position, e.g., as known from *feature-oriented programming (FOP)* [25, 7, 4], to combine separately modeled variability dimensions of the same concept.

- **Aggregation:** We support aggregation of variability model *instances* to be able to create complex models, e.g., representing a set of interdependent SPLs (i.e. multi product lines [28]).

In the following, we describe all three composition mechanisms in detail.

3.2.1 Inheritance

We use inheritance to extend and compose variability models. With single inheritance, we can extend an existing model to add new features and constraints. With multiple inheritance, we can furthermore compose multiple inherited concepts. In Figures 5 and 6, we show an example for FAMEDBMS. We describe operating system and build system specific variability in models OS and BUILD. We extracted the OS features from FAMEDBMS (cf. Figure 4) into the OS model (line 9 in Figure 6). FAMEDBMS inherits from both models (line 15) and thus merges the features and constraints of OS and BUILD. Furthermore, FAMEDBMS extends the merged models by adding new features and constraints (lines 16–20 in Figure 5).

Inheritance requires to merge one or more base feature models with the inheriting model. Segura et al. [31] and Acher et al. [2] describe how to merge feature models that represent an overlapping set of products into a single model, which represents the union of the products. In contrast to these approaches, we have to merge variability models that represent a different partial view on the variability of a single SPL. Hence, we aim at deriving (1) the union of all features and (2) the union of all constraints. The first operation increases variability by allowing all feature combinations of

```

1 concept Build {
2   mandatory feature Build {
3     mandatory feature Debugging { ... }
4     mandatory feature Compiler { ... }
5     mandatory feature System { ... }
6   }
7 }

```

```

8 concept OS {
9   mandatory feature OS {
10    oneOf {
11      feature Win32; feature MacOS; feature Linux;
12    }
13  }
14 }

```

```

15 concept FameDbms : Build, OS {
16   mandatory feature Access { ... }
17   ... // other domain-specific features
18
19   constraint win = OS.Win32 -> !Access.QueryEngine;
20   constraint !OS.MacOS;
21 }

```

Figure 6: Composition of variability models with inheritance.

the merged models. The second operation limits variability by joining the constraints of the models using conjunctions.

The merging can be achieved using the propositional representation of the feature model, as proposed by Czarnecki et al. [14]. That is, we translate the models into their propositional formula, merge these representations, and create a new feature model from the merged formula. Unfortunately, this may change the feature hierarchy, which can be a problem for some use-cases (e.g., for visualization). We thus propose to extend the approach to ensure that the parent-child relationship of the features is preserved. To solve naming conflicts (two features have the same fully qualified name but mean different things), features have to be renamed before composition as proposed in [2].

Constraints do not only exist within a single dimension but also between dimensions. That is, choosing a feature of one dimension may influence other dimensions. Hence, the constraints in a composed model can refer to the features of all merged concepts. For example, the constraint in line 19 of Figure 6 means that feature ACCESS.QUERYENGINE *must not* be selected when operating system WIN32 is selected.

Corresponding to the inheritance relationship between variability models, we define a subtype relationship between the models. Subtyping is required to support polymorphism when a client application uses multiple products of the same SPL [29] and for composition of larger SPLs [28], as we describe for aggregation below. We discuss subtyping in more detail in Section 4.

3.2.2 Superimposition

Another mechanism to compose variability dimensions is superimposition, as known from FOP. Superimposition enables domain engineers to decompose a concept into multiple variability dimensions without defining new concept names for the extracted dimensions. For example, the constraint in line 19 of Figure 6 is actually an implementation constraint. We thus may decompose the FAMEDBMS model into a domain model and an implementation model. The separated

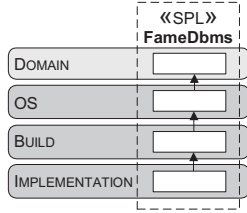


Figure 7: Decomposition of a variability model along variability dimensions (shown as layers).

```

Concept FAMEDBMS
1 concept FameDbms {
2   mandatory feature Access {
3     feature Logging { ... }
4   }
5   ...
6 }

Dimension BUILD
7 refines concept FameDbms {
8   mandatory feature Build { ... }
9 }

Dimension OS
10 refines concept FameDbms {
11   mandatory feature OS { ... }
12   constraint !OS.MacOS;
13 }

Dimension IMPLEMENTATION
14 refines concept FameDbms {
15   feature Access.Logging {
16     feature dLinux; //linux-specific implementation
17     constraint OS.Linux -> dLinux;
18
19     feature dWin32; //win32-specific implementation
20     constraint OS.Win32 -> dWin32;
21   }
22   constraint win = OS.Win32 -> !Access.QueryEngine;
23 }

```

Figure 8: Composition of variability models using superimposition.

dimensions can be superimposed with an external composition mechanism (e.g., using *FeatureHouse* [4]) without defining an inheritance relationship. Hence, superimposition is a scalable mechanism for composition of variability dimensions that avoids the combinatorial explosion when using inheritance.

In Figures 7 and 8, we depict a decomposition of FAMEDBMS based on superimposition of variability dimensions (shown as layers). Compared to Figure 5, we replaced inheritance with superimposition and added dimension IMPLEMENTATION. Dimensions OS (lines 10–13 in Figure 8) and IMPLEMENTATION (lines 14–23) include the constraints from lines 19–20 of Figure 6 and thus separate the implementation and operating-system-specific variability from the domain model of FAMEDBMS. Optional keyword `refines` (also used in FOP [7]) indicates that a model is an extension of a base model. We use the keyword to avoid accidental merging of models without a base model.

In the implementation model in Figure 8, we extend feature LOGGING with two alternative implementation features (DLINUX and DWIN32) for operating systems LINUX and WIN32 (lines 16 and 19). We furthermore add implementa-

tion constraints (lines 17 and 20) to define which implementation has to be used. Liu et al. described how such *feature interactions* (a.k.a. *derivatives*) can be explicitly modeled as separate interaction modules in a *concrete feature model* that represents the implementation view of an SPL [22]. Using superimposition, we are able to describe implementation variability separately from the implementation-independent domain model. We thus model the additional features only and avoid repeating the domain features.

For extending a base model with new features (e.g., extending ACCESS.LOGGING in line 15 of Figure 8), we provide a short notation that uses the fully qualified feature name (i.e., the list of parent features separated by a dot). In isolation (i.e., without a base model), the short notation corresponds to a chain of optional features. We can superimpose the optional features with any base model that includes without changing the relations between the features of the base model. Hence, the definition in line 15 means that features ACCESS and LOGGING are optional, whereas ACCESS is mandatory in the base model (line 2). Superimposition of both models preserves the original definition of ACCESS as a mandatory feature.

To understand how models with different relations between features are merged, we describe a part of the merge process in more detail. As already discussed, we may use a merge algorithm that is based on the propositional formula of the models. The definition of ACCESS.LOGGING in line 15 means that both features are optional. This is the least possible variability limitation. It can be represented with constraint: $Logging \Rightarrow Access$.

In contrast, feature ACCESS is mandatory in the base model (line 2), which can be represented with constraint (without simplification): $Access \wedge (Logging \Rightarrow Access)$.

Superimposition of both definitions via conjunction results in the latter constraint. Hence, the feature definition of the base model is preserved, because it is more restrictive than the optional feature of the superimposed model. In general, the more restrictive form is always preserved during a merge, because existing constraints are never removed.

An implementation model may have a completely different structure than the domain model. For example, when implementing an SPL with components, there is usually an m-to-n mapping of domain features to implementation features (i.e., the components). Hence, there are only a few overlapping features between the domain and the implementation model. The mapping of domain features to the implementation features can be described with constraints, as shown in Figure 8.

3.2.3 Aggregation

The last composition mechanism is aggregation, as known from OOP. We use aggregation to build larger SPLs by composing multiple SPL instances [28]. In contrast to inheritance, aggregation is based on SPL instances that represent concrete products. Aggregation allows an SPL engineer to compose instances of different SPLs and also differently configured instances of the same SPL.

We depict an example for aggregation in Figure 9. HYBRIDDB is an SPL for stream processing and persistent data storage. It makes use of two instances of FAMEDBMS (lines 5 and 9). Both instances are defined within a feature, which means that the instance is only required if the parent feature is selected. This is similar to *feature model references* [13]. We describe the dependencies between an SPL and the SPL

Concept HYBRIDDB

```

1 concept HybridDB {
2   feature ... //HybridDB features
3
4   feature Stream {
5     FameDbms streamDB;
6     constraint streamDB.Stream;
7   }
8   feature Persistent {
9     FameDbms persistentDB;
10    constraint persistentDB.Storage.Persistent;
11  }
12 }

```

Figure 9: Composition of SPL instances using aggregation.

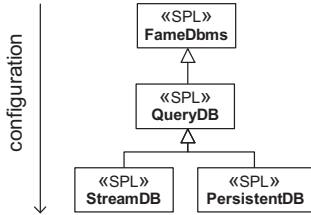


Figure 10: Specialization hierarchy of variability models.

instances it aggregates with *instance constraints* [28]. An instance constraint is shown in line 6 of Figure 9. The constraint is only valid for the SPL instance that is referenced in the constraint. Hence, the constraint in line 6 is only valid for instance `streamDB` and does not apply to instance `persistentDB`.

Instance definitions usually represent implementation details of an SPL. For example, `HybridDB` is implemented by two different variants of `FAMEDBMS`. Hence, the instance definitions may be separated from the domain model in an implementation dimension using superimposition. We discuss the combination of different mechanisms in Section 4.

3.3 Staged Configuration

We include staged configuration into variability modeling using *SPL specialization hierarchies* [29]. In Figures 10 and 11, we show the UML representation and the source code of a specialization hierarchy of the `FAMEDBMS` model. We use inheritance to represent SPL specialization by extending an unspecialized SPL. Each child in the hierarchy is a specialization of its parent (i.e., it represents less products), which we express by adding arbitrary *configuration constraints*. For example, we add a constraint in line 2 in Figure 11 to represent selection of feature `ACCESS.QUERYENGINE`. Hence, the feature is included in all products that can be derived from `QUERYDB`. Specializations `STREAMDB` and `PERSISTENTDB` further specialize `QUERYDB`. We can use the specialized SPLs also in instance definitions. For example, instead of the unspecialized `FAMEDBMS` (lines 5 and 9 in Figure 9), we can use an instance of `STREAMDB` and `PERSISTENTDB`.

3.4 Commutativity

A concept in `VELVET` is defined by a feature tree, constraints, and SPL instance definitions. Composition of concepts is commutative because features, constraints, and in-

Specialization QUERYDB

```

1 concept QueryDB : FameDbms {
2   constraint Access.QueryEngine;
3 }

```

Specialization STREAMDB

```

4 concept StreamDB : QueryDB {
5   constraint Stream;
6   constraint Storage.Inmem;
7   constraint !Access.Transaction;
8 }

```

Specialization PERSISTENTDB

```

9 concept PersistentDB : QueryDB {
10  constraint Storage.Persistent;
11 }

```

Figure 11: SPL specialization using inheritance and constraints.

stance variables are unordered and their composition is also commutative:

- Composition of the features of two SPLs means to superimpose their feature definitions. Changing the composition order thus only changes the order in which the (child)features are listed within the tree. This, however, does not change the semantics of the model.
- Constraint composition is commutative since we join constraints as conjunctions, which is commutative.
- Composition of SPL instance variables is also commutative, because it only changes the order in which the instances are listed.

This results in commutativity of the whole composition process. We can thus superimpose variability models in arbitrary order, which is beneficial for independent composition and configuration steps. Furthermore, when using multiple inheritance the composition order of the inherited concepts does not matter.

4. COMPARISON AND APPLICATION OF LANGUAGE MECHANISMS

`VELVET` integrates variability modeling, composition of variability models, and staged configuration. In the following, we compare the mechanisms and outline the possibilities the introduced concepts provide.

4.1 Comparison of Composition Mechanisms

The presented mechanisms for variability composition have benefits and drawbacks. We compare the mechanisms with each other to illustrate their applicability. An overview of the comparison is shown in Table 1.

Aggregation vs. Inheritance. Inheritance and aggregation are similar concepts and can often be used interchangeably [19]. In contrast to inheritance, aggregation is based on SPL *instances*. It also allows a domain engineer to combine multiple instances of the *same* SPL. We observed that this is needed when a client uses different variants of a component, each derived from a common code base [29]. In contrast, inheritance is used to extend and compose existing feature models and provides a subtype relationship corresponding to the inheritance relationship.

Mechanism	Usage	Objective
Inheritance	Model extension and composition	Reuse existing variability models
Superimposition	Merge different dimensions of an SPL	Independent developm. and configuration of variability dimensions
Aggregation	Aggregate SPL instances	Composition of <i>multi product lines</i>

Table 1: Application of mechanisms for variability composition

Superimposition vs. Inheritance and Aggregation. Superimposition can be used to freely combine different variability dimensions on demand. With inheritance or aggregation, we have to create a new concept when composing existing concepts, which does not scale.³ Furthermore, superimposition can be used to extend an existing feature model without invasive modifications. It is thus well suited for decomposing a variability model along different variability dimensions. In contrast, inheritance and aggregation are more appropriate when composing different variability models (with different names) that can be reused across SPLs. This is problematic with superimposition, which requires that the composed concepts have the same name.

4.2 Combining Different Mechanisms

By combining different mechanisms, we provide clean separation of variability dimensions and achieve high reuse at the same time.

Inheritance and Superimposition. In Figure 12, we show an example that combines inheritance and superimposition to add reusable variability models (OS and BUILD) to FAMEDBMS. We separate the inheritance definitions from the domain dimension into distinct modules (shown as white boxes within FAMEDBMS) in variability dimensions OS and BUILD (shown as layers). Within the modules, we can add FAMEDBMS-specific constraints for the composed models. For example, constraint `win` in line 22 of Figure 8 represents a dependency between domain model and chosen operating system. Defining the constraint in a separate dimension of FAMEDBMS (layer OS in Figure 12) separates it from the FAMEDBMS domain model and also from the reusable OS variability model. The same can be done for aggregation. Hence, with inheritance we compose reusable variability models and with superimposition we overcome the scalability limitations of inheritance.

Specialization and Generalization. Before using a feature model for generalization, we may specialize it to omit variability not needed for a concrete SPL. For example, a concrete DBMS SPL will not cover the entire DBMS domain. By specializing a DBMS feature model, which describes the entire domain, we can derive a model for a subdomain that is covered by the concrete SPL. This specialized model may be extended again with features specific for the subdomain.

Specialization and Superimposition. Finally, we can modularize specialization steps in separate configuration

³The number of required concepts increases exponentially for an increasing number of variability dimensions.

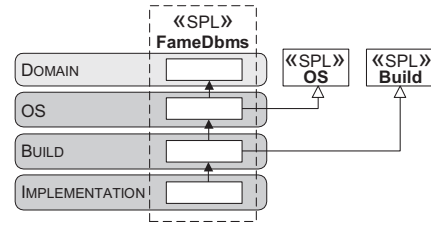


Figure 12: Combining inheritance (empty arrow head) and superimposition (filled arrow head) to separate and reuse variability dimensions.

models and combine the separated models with superimposition. With alternative or optional superimpositions, we can provide a customizable configuration process. This allows us to change the configuration of a specialized SPL, a set of specializations (i.e., a specialization hierarchy), or a number of different SPLs without invasive modifications. For example, when we want to choose between different build settings (DebugBuild vs. TestBuild vs. ReleaseBuild) for a specialization hierarchy (e.g., as shown in Figure 10), we can modify specializations of the hierarchy by applying selected superimpositions without invasively modifying the existing hierarchy. This can also be applied to provide alternative specializations for different customers or application scenarios. In this case, each feature of this higher level variability model groups a set of configuration choices (e.g., for a particular customer or application scenario).

4.3 Subtyping and Polymorphism

In previous work, we defined a subtype relationship between an SPL and its specializations [28]. For example, concept FAMEDBMS in Figure 5 is a subtype of concept BUILD. We now extend the subtype relationship to include arbitrary SPL extensions (i.e., generalization and specialization). For SPL specializations, we defined that a subtype has to provide all *bound* features of its super type and may bind further features. We extend this for variability modeling in general and define that a subtype (1) has to provide *all* features (*bound* plus *unbound*) of its supertype and (2) all features that are bound in the supertype also have to be bound in the subtype. A subtype may add further features (variability extension) and bind existing or added features (specialization).

Consequently, we can use SPLs polymorphically: we can provide subtype P' at positions where its supertype P is required because the subtype provides a superset of all features and a superset of all bound features of its supertype. This is needed for aggregation, which does not require to define the concrete type of an SPL but only a supertype that defines the minimally required features [29]. Concrete SPL instances are created by a composition program as we outlined in [28]. However, such a composition program has to consider subtyping relationships and constraints for a set of SPL instances, which we intend to address in future work.

With subtyping and tool support, it is possible to detect errors also in complex multi product lines that consist of several SPLs. For example, we can detect errors without knowing the concrete configuration of a referenced SPL instance by checking if the instance is a subtype of the required type.

4.4 Overlapping Variability Dimensions

As already discussed, features may overlap when decomposing a variability model into multiple dimensions. With the presented compositional approach, we can model overlapping dimensions separately. However, this results in repeating the overlapping features. There are several solutions that avoid or reduce replication of features:

- One of two overlapping dimensions is modeled as an extension of the other dimension. When adding a child feature in the extending model to an already existing parent feature, only the path to the parent has to be repeated.
- The overlapping dimensions may be further decomposed to extract an additional dimension that captures the commonalities (i.e., the overlapping features).

The applicability and drawbacks of both solutions and their combination have to be further analyzed. Similar problems also occur in FOP and may be solved with a combination of annotative and compositional approaches [21].

4.5 Variability Analysis and Visualization

Separate modeling of variability dimensions allows a domain engineer to analyze and visualize the dimensions separately or in combination. For this purpose, we can use a projection of a subset of all dimensions of the multi-dimensional variability space. For example, operating-system-specific variability of an implementation can be analyzed by choosing dimensions IMPLEMENTATION and OS only. In such a *view* on selected variability dimensions, we may highlight individual dimensions, overlapping features, and interactions between dimensions.

How to visualize and analyze composed variability models requires further research. For example, it is possible to visualize inheritance between a base model and an extending model as two distinct models or by superimposing them (e.g., after removing dead features that have been introduced due to added constraints). The same applies to aggregation, which can be shown by integrating aggregated variability models or by using feature model references. Based on the presented language, it is possible to build tools for analysis and visualization of feature models.

5. RELATED WORK

Variability Modeling Languages. There are several textual languages for variability modeling, such as Clafer [5], FDL (feature description language) [15], guidsl [6], TVL (text-based variability language) [9], and VSL (variability specification language) [1]. VELVET is a generalization of single-dimensional approaches for variability modeling but still allows SPL engineers to use only a single dimension. In this case, VELVET does not provide a higher complexity than existing languages. In contrast to other languages, TVL and VSL support variability composition. Composition in TVL is similar to our aggregation mechanism. VSL furthermore supports inheritance. We integrate several mechanisms in a single language and support subtyping between variability models. Moreover, we integrate variability modeling and staged configuration. However, we do currently not support cardinality-based feature modeling.

Variability composition and decomposition. Czarnecki et al. use cardinality-based feature models with *feature*

model references and *feature cloning* to compose variability models [13, 20]. The tool *FeaturePlugin* allows for configuring multiple feature models that are integrated via feature model references [3]. Hartmann et al. propose to compose feature models of different suppliers to derive a supplier independent feature model. Feature model references are similar to the aggregation mechanism we use for model composition. We additionally use instance constraints to describe the dependencies between an SPL and aggregated instances.

Reiser et al. describe a way to manage complex product lines with *multi-level feature trees* [27], which are implemented in the CVM framework and VSL [1]. They propose to use *reference feature models* to describe the variability that is common for a set of related feature models. A *referring* feature model adds or restricts variability of its reference model. This approach is similar to our inheritance-based composition. The reference feature model thus corresponds to the extended model in an inheritance relationship. We also support multiple inheritance to compose feature models and provide additional composition mechanisms.

Thompson et al. introduced *n-dimensional* program families [35]. They structure SPLs by describing their commonalities and variabilities along multiple dimensions based on set theory. This is similar to our approach. However, they do not propose a language for variability modeling nor do they include the configuration process.

Segura et al. use graph transformations for merging feature models of different SPLs [31]. They provide a set of rules to merge the feature models. Similarly, Acher et al. provide different ways to compose the feature models of multiple SPLs [2]. Both approaches aim at merging feature models that represent a potentially overlapping set of products to yield the union of the products. We merge different variability dimensions representing a subset of the variability of a single SPL. We thus use a different merge algorithm. The approach of Acher et al. is an alternative to aggregation of features models with references as we do. In contrast to a reference-based solution, their approach yields more compact feature models when there are overlapping features. Hence, it may be combined with our approach.

Metzger et al. describe *product line variability* and *software variability* in separate models and define dependencies with cross-model links [23]. Streitferdt et al. integrate feature models of *hardware* and *software* product lines to derive the hardware configuration via model dependencies [33]. Similarly, the tool `pure::variants`⁴ allows to define implementation variability separately from the domain model. Separating domain and implementation variability is a special case of multi-dimensional separation, as we proposed.

Hubaux et al. propose views on feature models, e.g., to manage access rights [17]. They do not decompose a feature model into separate dimensions but define which features are part of a particular dimension. Hence, it is an annotative approach while we use a compositional approach.

Rabiser et al. [26] propose to adapt and augment variability models to include additional information needed for product derivation. In contrast to our approach, they focus on creating a derivation model from variability models by adding and removing information. Similarly, we and others extended feature models with additional attributes required for automated product derivation (e.g., [32, 8, 38]). With

⁴http://www.pure-systems.com/pure_variants.49.0.html

VELVET, we aim at describing variability dimensions independently to compose them as needed. Due to support for attributes, it is also possible to describe additional information (e.g., non-functional properties) in separate dimensions, which can be used for automated product derivation [32].

Fries et al. present an approach to model compositions of multiple SPLs [16]. They use *feature configurations*, a selection of features, to describe a group of SPL instances that share these features. Feature configurations are similar to SPL specializations but do not allow a user to describe multiple configuration steps. We use a specialization hierarchy to describe the commonalities of a set of specialized SPLs and reuse configuration decisions. With superimposition, we can apply independently defined feature configurations to an inheritance hierarchy.

Staged Configuration. Czarnecki et al. [12] and Classen et al. [10] define staged configuration as a process that has to eliminate configuration choices. We do not adhere to this strict definition. A configuration step may be defined as a separate module that defines configuration constraints. Automatically applying multiple configurations may thus result in configuration steps that do not reduce the variability because the variation points have already been bound before. Nevertheless, it can be easily checked whether a configuration step reduces the variability by using a SAT solver [36].

We define a configuration step as a set of constraints over the features of an SPL. This was also described by Classen et al. [10]. In contrast to the specialization steps defined by Czarnecki et al. [12], we support arbitrary configuration constraints and achieve commutativity of configuration steps. This allows us to apply specializations in arbitrary order.

6. CONCLUSION

Variability of a software product line (SPL) is sometimes described in multiple separate variability models and sometimes described in a single complex model. Both ways have benefits and drawbacks with respect to communication between stakeholders and the SPL development process. Furthermore, SPL configuration and variability modeling are usually considered independently even though both use similar concepts.

We presented VELVET, a language that integrates separate modeling of variability dimensions, composition of variability models, and SPL configuration. VELVET allows a stakeholder to decompose the variability model of an SPL into multiple models to handle its complexity. Models can be composed using three different mechanisms: inheritance, superimposition, and aggregation. By combining the mechanisms, we can reuse variability models across SPLs while separating the variability dimensions of a single SPL. Finally, we have shown that SPL configuration (i.e., specialization) is a mechanism for variability reduction, which we can describe in VELVET with the same concepts that we use for variability modeling.

Many of the mechanism we use in VELVET have already been proposed in a similar form; we now integrate them with a unified concept for variability modeling. In future work, we plan to provide tool support for VELVET (e.g., as part of FeatureIDE⁵) and to evaluate it with a case study. We also aim at exploring the properties of feature attributes and the applicability of other OOP constructs.

⁵<http://fosd.de/featureide>

Acknowledgments

We thank Sven Apel for comments on earlier drafts of this paper. Marko Rosenmüller is funded by German Research Foundation (DFG), project number SA 465/34-1.⁶ Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C.⁷

7. REFERENCES

- [1] A. Abele, R. Johansson, H. Lönn, Y. Papadopoulos, M.-O. Reiser, D. Servat, M. Törngren, and M. Weber. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 101–105, 2010.
- [2] M. Acher, P. Collet, P. Lahire, and R. France. Composing Feature Models. In *Int'l. Conf. Software Language Engineering (SLE)*, volume 5969 of *LNCS*, pages 62–81. Springer, 2009.
- [3] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In *Eclipse '04: Proc. 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72. ACM Press, 2004.
- [4] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 221–231. IEEE CS, 2009.
- [5] K. Bak, K. Czarnecki, and A. Wasowski. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *Int'l. Conf. Software Language Engineering*, 2010.
- [6] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
- [7] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng. (TSE)*, 30(6):355–371, 2004.
- [8] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *Int'l. Conf. Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005.
- [9] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-based Feature Modelling. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 159–162, 2010.
- [10] A. Classen, A. Hubaux, and P. Heymans. A Formal Semantics for Multi-level Staged Configuration. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 51–60, 2009.
- [11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [12] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-level

⁶<http://fosd.de/multiple>

⁷<http://vierfores.de>

- Configuration of Feature Models. In *Software Process Improvement and Practice 10*, pages 143–169, 2005.
- [13] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, volume 3154 of *LNCS*, pages 266–283. Springer, 2004.
- [14] K. Czarnecki and A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 23–34. IEEE CS, 2007.
- [15] A. Deursen and P. Klint. Domain-specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.
- [16] W. Friess, J. Sincero, and W. Schroeder-Preikschat. Modelling Compositions of Modular Embedded Software Product Lines. In *Proc. 25th Conf. IASTED Int'l. Multi-Conf.*, pages 224–228. ACTA Press, 2007.
- [17] A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards Multi-view Feature-Based Configuration. In *Int'l. Working Conf. Requirements Engineering: Foundation for Software Quality*, volume 6182 of *LNCS*, pages 106–112. Springer, 2010.
- [18] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [19] H. Kegel and F. Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 431–440. ACM Press, 2008.
- [20] C. H. P. Kim and K. Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *Europ. Conf. Model Driven Architecture Foundations and Applications (ECMDA)*, pages 331–348, 2005.
- [21] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. Int'l. Conf. Generative Programming and Component Eng. (GPCE)*, pages 157–166. ACM Press, 2009.
- [22] J. Liu, D. Batory, and S. Nedunuri. Modeling Interactions in Feature-Oriented Designs. In *Proc. Int'l. Conf. Feature Interactions (ICFI)*, pages 178–197. IOS Press, 2005.
- [23] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Int'l. Requirements Engineering Conf. (RE)*, pages 243–253. IEEE CS, 2007.
- [24] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [25] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.
- [26] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting Product Derivation by Adapting and Augmenting Variability Models. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 141–150. IEEE CS, 2007.
- [27] M.-O. Reiser and M. Weber. Managing Highly Complex Product Families with Multi-Level Feature Trees. In *Proc. Int'l. Conf. Requirements Engineering*, pages 146–155. IEEE CS, 2006.
- [28] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, 2010.
- [29] M. Rosenmüller, N. Siegmund, and M. Kuhlemann. Improving Reuse of Component Families by Generating Component Hierarchies. In *GPCE Workshop on Feature-oriented Software Development (FOSD)*, pages 57–64. ACM Press, 2010.
- [30] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 1–6. School of Computer Science, University of Magdeburg, 2008.
- [31] S. Segura, D. Benavides, A. R. Cortés, and P. Trinidad. Automated Merging of Feature Models Using Graph Transformations. In *Int'l. Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 5235 of *LNCS*, pages 489–505. Springer, 2007.
- [32] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kästner, and G. Saake. Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 25–31, 2008.
- [33] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow. Configuring Embedded System Families Using Feature Models. In *Proc. of Net.ObjectDays*, pages 339–350. Gesellschaft für Informatik, 2005.
- [34] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 107–119. IEEE CS, 1999.
- [35] J. M. Thompson and M. P. E. Heimdahl. Extending the Product Family Approach to Support n-Dimensional and Hierarchical Product Lines. In *Int'l. Symposium on Requirements Engineering (RE)*, pages 56–65. IEEE CS, 2001.
- [36] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 254–264. IEEE CS, 2009.
- [37] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proc. Working Conf. Software Architecture (WICSA)*, pages 45–55. IEEE CS, 2001.
- [38] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating Product-Line Variant Selection for Mobile Devices. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 129–140. IEEE CS, 2007.