Optimizing Design for Variability Using Traceability Links

Matthias Riebisch, Robert Brcina
Technical University of Ilmenau, Germany
matthias.riebisch|robert.brcina@tu-ilmenau.de

Abstract

Software systems have to provide flexibility by implementing variability. Existing design methodologies do not support means for optimizing the design for variability and for measuring the overhead effort. Therefore, the solutions cannot be optimized regarding a minimal overhead for variability. Other methods are lacking of a traceability for variability mechanisms, or do not provide means for measuring and optimizing solutions. The paper introduces traceability links for variability with a special emphasis on support for implementation, build and deployment, and presents guidelines for optimizing the design with indicators for evaluating the results. The feasibility of the approach is shown by a case study from an industrial setting.

1 Introduction

Software systems in most domains have to provide a high capability for being changed, e.g. for mobile phones or business information systems. Changes are frequently caused by changed customer requirements or by evolving technical platforms and environments. Such a demand for flexibility and changes has lead to the development of software product lines [10]. A product line consists of a common core which is extended by components to configure new products quickly. Due to hard time and cost constraints there is a high pressure to use existing components from the market. These components are largely heterogeneous, e.g. from different providers or designed in different styles. Furthermore, most of these components are black boxes, which means that a developer can only access the interfaces and not the internal structure of a component. Usually, a component does not ideally fit to the features of a currently designed product, e.g. concerning its interfaces, its features or its structure. Therefore, the configuration of software systems constitutes a challenging and success-critical task. A configuration activity is especially critical, if it was not intended by design. Problems can be caused by e.g. missing variability points (sometimes called hot spots) of components or unknown side effects. If such problems are not solved in the architectural design, they have negative effects on the quality attributes of a system. Furthermore, the high complexity of the configuration task increases the probability of mistakes, especially in the case of large systems with a high number of components and constraints between them. Different versions of the components increase the complexity of the configuration task.

Following the ideas of model-driven development, we have developed a method for supporting the activities of configuration and deployment by utilizing traceability between requirements, features and implementation artefacts. The method helps to resolve variability constraints introduced by third party components. It facilitates the system configuration by providing a better guidance and tool support. In this paper, we apply traceability links to map features to implementation components and artefacts – used during build and deployment activities. Using these links we evaluate constraints between artefacts of each phase and map them to the feature model. They facilitate the feature configuration, the architectural design driven by features, and the component implementation. In this way possible conflicts for the configuration activities are represented in models. By applying rules for a good design, the method uses indicators and evaluations to provide guidance for improving development activities.

2 Build and Deployment Related Design Goals

We want to support the design process in terms of variability, with a special emphasis on support for build and deployment. Therefore, the general goals for a high-quality design had to be fulfilled. Firstly, a design method has to implement the functional requirements completely. Secondly, all variability and quality requirements have to be fulfilled by the resulting solution. Furthermore, all criteria for a good design have to be met, e.g. modularity, abstraction, encapsulation, separation of concerns and conceptual integrity. To achieve an efficient design process, the method has to possess quality attributes like simplicity, comprehen-

sibility, and tool independence.

In addition to the previous goals, the method has to lead to highly flexible solutions. High flexibility means *variability with a low extra effort*, both for achieving a set of options, and for adaptation by activating one option instead of a previous one. For all changes *regarding features* a minimal effort is achieved, if a change of a feature can be implemented by a pure insertion or substitution of a component, without any impact on other parts of a system. In terms of relations between features and implementation components an architecture with only 1:1 relations between them leads to a minimal effort and thus a maximal flexibility. In such an architecture, just component replacements without any additional code changes are necessary if features are changed. The architectural style of plug-in components provides one solution for an appropriate variability infrastructure for changes at start-up time or at runtime. Other examples for solutions for runtime variability are the design patterns *Strategy*, *Abstract Factory* or *Template Method* [7]. An example solution for variability at compile time is conditional compilation.

3 Traceability Approach Integrating Implementation Artefacts for Variability

Artefact Categories. For the design optimization concerning variability, the relations between requirements, architectural elements and implementation have to be considered. The key idea is to trace all software development artefacts back to its requirements. For the further investigation, the linked artefacts are grouped and categorized in levels. Within a feature driven development process for software product lines we have to consider the artefacts of the whole process. The levels are distinguished by different types of artefacts and dependencies between them. In the following we categorize them to a feature level (F-Level), architectural component level (A-Level), class level (C-Level) and implementation artefact level (I-Level). Examples for artefacts of the I-Level are configuration units or sections.

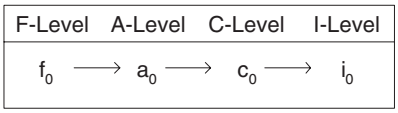


Figure 1. Ideal case of 1:1 relations

Typical effects. From a changeability and complexity point of view the relations in figure 1 represent an ideal case because 1:1 relations between the artefacts of the different levels lead to a minimal effort for changes. In the case of a change of feature f_0 only a change of the component a_0 is required. In this ideal case each artefact does not have more than two traceability links to other artefacts.

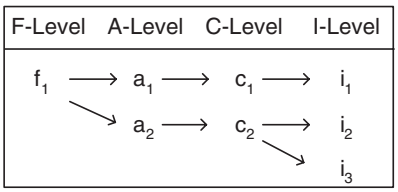


Figure 2. The case of feature scattering

In reality more dependencies have to be considered. A higher number of dependencies means that more artefacts are affected by a change resulting in a higher maintenance effort and a reduction of the variability. Two types of effects are discussed here. Feature scattering (figure 2) means that one feature f_1 is implemented by more than one architectural components – in this case the components a_1 and a_2. The addition of this feature to a product requires the integration of more than one component.

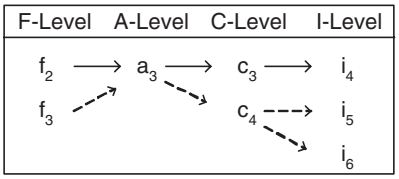


Figure 3. The case of feature tangling

In the case of feature tangling, an architectural component is responsible for more than one feature, (figure 3) e.g. the implementation of feature f_2 and f_3 is tangled in component a_3. If one of these features has to be removed, the component a_3 has to be analyzed and split into appropriate parts, with a much higher effort than just the removal of one component. From the variability point of view both cases should be avoided. As also stated in [14] feature tangling and scattering have negative impacts on the system's maintainability. In order to improve the flexibility of a system concerning feature variability, all variability points should be aligned in a way that each of them is related to exactly one optional feature. Tangling and scattering again leads to a higher number of dependencies which have to be considered during all kinds of change activities, like program comprehension or impact analysis.

In this paper, all artefacts with relevance to variability are considered, e.g. system components, classes and implementation artefacts. These relations enable an evaluation and optimization of the system architecture concerning variability. In the case of figure 3 the determination of the artefacts relevant for an implementation of the feature f_2 would not be possible without traceability links between i_4

to c_3 and f_2 (solid line). The same holds for the analysis of the impact of a change of feature f_3 (dashed line). Furthermore, a comprehensive analysis of artefacts from several categories and their relations is necessary in order to be able to evaluate the degree of variability, to forecast a possible configuration overhead, and to verify the consistency between feature model and architectural models.

Types of relations. It is necessary to describe additional relations between artefacts as e.g. the relation between classes and features and to determine several sets of artefacts used in the current traceability approach. For the illustration of this approach we will use a small example (figure 7) out of a Home Automation System (HAS) used in [10].

The set of features F and two considered subsets $\mathcal{F} \subseteq F$, $\mathcal{F}_o \subseteq F$ are all contained in feature models. Note that $\mathcal{F}_o \subseteq F$ is a set of optional features. The set of architectural software components A and the considered subset $\mathcal{A} \subseteq A$, part of the architecture models. The set of classes C and the considered subset $\mathcal{C} \subseteq C$, part of the realisation models. The set of implementation artefacts I and the considered subset $\mathcal{I} \subseteq I$, part of the realisation models. In the following the traceability links and relations are defined, which are required for the later evaluation by the indicators (see section 4):

Component Traceability. Each component contributes to a set of requirements. Such a relationship is expressed by the "implementedBy" traceability link pointing to components that implement a set of features.

Definition: Component Traceability
We use the symbol $f \rightsquigarrow \mathcal{A}$ as "implementedBy" traceability link expressed as follows:

$$f \rightsquigarrow \mathcal{A} :\Leftrightarrow f \in F, \mathcal{A} \subseteq A : implementedBy(f, \mathcal{A}). \quad (1)$$

If considering exactly one feature and one component the same traceability link type is used.

The number of components that implement the feature f is countable with $nco(f)$ which is defined as:

$$nco\,(f) := |\{a \in A : f \rightsquigarrow a\}|\,. \quad (2)$$

Component Require Relation. The "use" traceability link describes the relationship between two components in which one component needs the other component to implement the related feature. Formally, this is expressed by the following definition:

Definition: Component Require Relation
The symbol $a \mapsto \mathcal{A}$ expresses the "use" traceability link between source component a and a set of other components A, but a source component a may not use itself ($a \notin \mathcal{A}$).

$$a \mapsto \mathcal{A} :\Leftrightarrow a \in A, \mathcal{A} \subseteq A, a \notin \mathcal{A} : require(a, \mathcal{A}). \quad (3)$$

The same traceability link type is used if only a relationship between two components is considered.

Class Traceability. Software components consist of a set of classes and vice versa a class c is related to exactly one software component in order to implement at least one part of a feature. The same traceability link type (and also the same symbol) defined for components is also used for classes: $f \rightsquigarrow \mathcal{C}$. The number of classes that implement the feature f is countable with $ncl(f)$ and is defined as:

$$ncl\,(f) := |\{c \in C : f \rightsquigarrow c\}|. \quad (4)$$

Implementation Artefact Usage. It is necessary to trace interactions between classes and implementation artefacts within architectural components that are related to a feature. We denote an Implementation Artefact Usage if at least one class needs an implementation artefact in order to adapt aspects of the feature implementations.

Definition: Implementation Artefact Usage
In order to support the analysis of the usage of implementation artefacts by classes $\mathcal{C}$ we use the symbol $\mathcal{C} \longmapsto \mathcal{I}$ for this kind of traceability link type named as "use", formally expressed as follows:

$$c \longmapsto \mathcal{I} :\Leftrightarrow c \in C, \mathcal{I} \subseteq I : use(c, \mathcal{I}). \quad (5)$$

Implementation Artefact Traceability. An implementation artefact is usable for different purposes. In this paper an implementation artefact is a section of a configuration database or a node of a configuration tree in a configuration file (e.g. the structure known from a registry), which is used to adapt the feature, e.g. switching off a feature before the deployment of the component. Such a relationship is expressed by the "isAdaptedBy" traceability link:

Definition: Implementation Artefact Traceability
We use the symbol $f \overset{ia}{\rightarrow} \mathcal{I}$ as "isAdaptedBy" traceability link expressed as follows for $c \in \mathcal{C}$. Note that this definition considers two conditions, firstly classes must use an implementation artefact $\mathcal{C} \longmapsto \mathcal{I}$ and secondly, these classes are used to implement a feature $f \rightsquigarrow \mathcal{C}$. Only if these conditions are valid, an adaption of a feature is possible; this is expressed as follows:

$$f \overset{ia}{\rightarrow} \mathcal{I} :\Leftrightarrow f \in F, \mathcal{I} \subseteq I, f \rightsquigarrow \mathcal{C}, \quad (6)$$
$$\mathcal{C} \longmapsto \mathcal{I} : isAdaptedBy(f, \mathcal{I}).$$

If considering exactly one feature and one implementation artefact the same traceability link type is used.

The number of implementation artefacts that adapt the feature f is countable with $nia(f)$ and is defined as:

$$nia\,(f) := |\{i \in I : f \xrightarrow{ia} i\}|. \tag{7}$$

4 Indicators for Evaluation of the Goals

4.1 Feature Scattering

As discussed earlier in this paper feature scattering and tangling constitute major deficiencies with respect to software maintenance, variability or reducing configuration and implementation overhead. Before feature scattering can be evaluated, insulated features have to be resolved.

Insulated Features. A feature must have a traceability link to at least one architectural component. This is an important condition which contributes to the consistency between features and their related components. An example

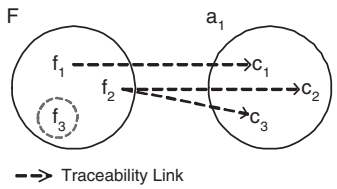


Figure 4. Example of an insulated feature

of an insulated feature f_3 is depicted schematically in figure 4. The indicator $if(F)$ is defined as follows:

Definition: Insulated Feature Indicator
Each feature f must be implemented by at least one component otherwise ($\nexists \mathcal{A}$) this feature is an insulated feature and countable with $if(F)$.

$$if(F) := |\{f \in F : \nexists \mathcal{A} \subseteq A : f \rightsquigarrow \mathcal{A}\}|\,. \tag{8}$$

From the engineering point of view insulated features could exist for several reasons: The feature's realisation is postponed, as during domain analysis not all information could be gathered. This often happens in the case of low priority features. Additionally, in the case of non-functional requirements it is often hard to determine how to map their implementation to elements of the existing system.

Resolution: All insulated features have to be resolved

A resolution step is necessary to remove an insulated feature by (a) postponing such features to later releases, by (b) mapping it to proper architectural components.

Feature Scattering. After removing all insulated features the feature scattering indicator is applicable. On the architectural level, feature scattering refers to a relation between one feature and more than one component. Feature scattering affects the maintenance or the implementation overhead of a system because it is not always easily possible to adapt the implementation of a feature in order to fulfill the customers' requirements considering variability. A high implementation overhead leads to higher effort and to a higher probability of mistakes.

Definition: Feature Scattering Indicator
The indicator $sca(f)$ is used to determine the number of components ($a \in A$) that are needed to implement the feature $f \in F$. Additionally, $fsca(F)$ is used to analyse all features and to provide an overall measurement result.

$$sca\,(f) := |\{a : f \rightsquigarrow a\}| - 1. \tag{9}$$

Note that $sca(f)$ considers the ideal case where no feature scattering exists (by the subtraction of 1).

$$fsca\,(F) := \frac{\sum_{f \in F} sca\,(f)}{|F| \cdot |A|},\ fsca \in [0, 1)\,. \tag{10}$$

The more features are scattered into components, the worse the maintainability of the software gets and the closer the result of the indicator moves to 1. The maximum value 1 is reached if $|a|$ approaches infinity and each feature f is implemented by all a. For figure 4 $fsca(F)$ is $1/6$, if the insulated feature f_3 is resolved before ($|A| = 3$, $|F| = 2$).

Resolution: Reducing feature scattering

Reducing feature scattering could be done by (a) splitting up the features into several features starting with the feature with the highest $sca\,(f)$ value, (b) try to merge components reducing the number of involved components.

4.2 Feature Tangling

Insulated Components. Similar to section 4.1 we identify the removal of insulated components $a \in A$ as a precondition for the use of the Feature Tangling Indicator. More precisely, at least one traceability link must exist from a feature f pointing to an existing architectural component $a \in A$. It is important to achieve a complete realisation in order to ensure the consistent evaluation of the system's engineering approach. An insulated component has a similar structure as an insulated feature; this is already depicted schematically in figure 4.

We call the indicator for such a component situation Insulated Component Indicator and define it as follows:

Definition: Insulated Component Indicator

$$co(A) := |\{a \in A : \nexists f \subseteq F : f \rightsquigarrow a\}| . \quad (11)$$

From the engineering point of view, insulated components constitute a mistake. Nevertheless, they could occur if e.g. an architecture contains components without a direct relation to customer needs.

Resolution: All insulated components must be resolved

A resolution step is necessary in order to remove the insulated component e.g. by (a) creating a dummy feature, by (b) introducing internal features or by (c) removing the insulated component. This step helps to avoid or at least to reduce the quite common effect so-called "Bells and Whistles" features, which are implemented by developers but are not wanted or paid for by customers.

Feature Tangling. Feature tangling refers to relations between one feature and more than one component.

Definition: Feature Tangling Indicator
The indicator $tang(a)$ is used to determine the number of features $f \in F$ that are implemented by a component $c \in C$. Additionally, $ftang(A)$ is used to analyse all features and to provide an overall measurement result.

$$tang\,(a) := |\{f : f \rightsquigarrow a\}| - 1. \quad (12)$$

Note that $tang(a)$ considers the ideal case where no feature tangling exists (by the subtraction of 1).

$$ftang\,(A) := \frac{\sum_{a \in A} tang\,(a)}{|F| \cdot |A|}, ftang \in [0, 1) . \quad (13)$$

The more features are tangled to one component, the more difficult is the adaption of this component and the closer is the result of the indicator to 1. The maximum value 1 is reached if $|f|$ approaches infinity and each component $a \in A$ implements all features $f \in F$.

Resolution: Reducing feature tangling

Reducing the relations between more than one feature and a component could be done by (a) splitting up the component starting with the highest number of $tang\,(a)$ into several components, by (b) trying to merge features in order to reduce the number of features involved.

4.3 Design of Implementation Artefacts

As discussed earlier in this paper feature scattering and tangling constitute major deficiencies with respect to variability or the configuration overhead. Implementation artefacts have the role of endpoints of a feature's realisation as they are often used to adjust the implementation, e.g. to switch off or on a feature's implementation in relation to its defined variation points in the feature model. Additionally, if a lot of the feature's parameters have to be adapted e.g. time restrictions that are scattered to several configuration locations (we call them implementation artefacts) typical activities like testing or deployment often fail. Especially within the customer's environments such a problem situation requires expensive development support. The reason for such an expensive support is quite simple, important settings are overseen by the engineer due to scattered configuration units and settings.

Insulated Implementation Artefacts. Before Implementation Artefact Scattering or Tangling can be evaluated, insulated features (already discussed in section 4.1) and Insulated Implementation Artefacts have to be resolved. More precisely, an implementation artefact must have a direct relation to at least one class whereas this class is linked by a traceability link to at least one feature as well. This is an important condition which contributes to the consistency between features and their related implementation artefacts. An insulated implementation artefact has a similar structure as an insulated feature, which is already shown in figure 4.

Definition: Insulated Implementation Artefact Indicator

$$ia(I) := |\{i \in I : \nexists f \subseteq F : f \overset{ia}{\rightarrow} i\}|. \quad (14)$$

From the engineering point of view an insulated implementation artefact could exist in the case of dead code in which the implementation artefact still has no role anymore. Such a situation occurs if the feature's realisation is postponed, as during the domain analysis phase not all information are collected, but the prototype realisation still exists.

Resolution: All cases of insulated implementation artefacts must be resolved

A resolution step is necessary in order to remove an insulated implementation artefacts for example by (a) removing such dead code, by (b) reawaken such an implementation artefact through mapping it to proper architectural components and the related feature.

Implementation Artefact Scattering. We consider situations in which features have to be configured by several implementation artefacts, but hamper the activities during deployment and error diagnostics of an application. This is often the case e.g. in a configuration tree (xml-configuration files) in which each configuration unit is scattered to several not grouped sections, locations or nodes.

The misconfiguration of feature related runtime parameters is quite a serious problem, as often a program error occurs only at specific runtime situations and requires high analysis efforts in the customer's environment. Such an expensive problem situation is often caused due to a lack of traceability. The Implementation Artefact Scattering Indicator is used to evaluate the realisation in order to achieve an effective configuration of the system. An example of implementation artefact scattering is shown in figure 7, the configuration of feature $Control\ Center$ is scattered into implementation artefact $Configuration$ and $Settings$.

Definition: Implementation Artefact Scattering Indicator
The indicator $iasca(f)$ is used to determine the number of implementation artefact $i \in I$ that are used to adapt the feature $f \in F$. Additionally, $fiasca(F)$ is used to analyse all features and to provide an overall measurement result.

$$iasca\,(f) := |\{i : f \overset{ia}{\rightarrow} i\}| - 1. \tag{15}$$

Note that $iasca(f)$ considers the ideal case where no implementation artefact scattering exists (by the subtraction of 1).

$$fiasca\,(F) := \frac{\sum\limits_{f \in F} iasca\,(f)}{|F| \cdot |I|},\ fiasca \in [0, 1)\,. \tag{16}$$

For the development of systems concerning variability it is important to pay attention to the design and to reduce the number of possible misconfigurations.

Resolution: Reducing implementation artefact acattering

Reducing implementation artefact scattering could be done by (a) splitting up the features into several features starting with the feature with the highest $iasca\,(f)$ value, (b) trying to merge or group implementation artefacts in order to reduce the number of implementation artefacts involved in different sections.

There are cases where a resolution is not possible if e.g. external components or products are used. In these cases the role of an architectural decision is important. Because it documents alternatives and is also the base for an appropriate installation and deployment documentation.

Implementation Artefact Tangling. We identify the removal of insulated features in section 4.1 and insulated implementation artefacts in section 4.3 as a precondition for the use of the Implementation Artefact Tangling Indicator.

Situations in which features have to be adapted by the same implementation artefact hamper implementation and deployment activities and error diagnostics as it is not easy to keep track of the used variation points in the configuration file. Additionally, it is often a symptom indicating less flexibility considering variability as it is not possible to plug out a component with its implementation artefact as it contains the configuration of other features, too. An example of implementation artefact tangling is shown in figure 7, the features $Control\ Center$ and $Modul\ Management$ are adaptable by the same implementation artefact $Configuration$.

Definition: Implementation Artefact Tangling Indicator
The indicator $iatan(i)$ is used to determine the number of features $f \in F$ that are adapted by one implementation artefact $i \in I$. Additionally, $fiatan(I)$ is used to analyse all implementation artefacts and to provide an overall measurement result.

$$iatan\,(i) := |\{f : f \overset{ia}{\rightarrow} i\}| - 1. \tag{17}$$

Note that $iatan(i)$ considers the ideal case where no implementation artefact tangling exists (by subtraction of 1).

$$fiatan\,(I) := \frac{\sum\limits_{i \in I} iatan\,(i)}{|F| \cdot |I|},\ fiatan \in [0, 1)\,. \tag{18}$$

Resolution: Implementation artefact tangling

Reducing the relations between more than one feature and one implementation artefact could be done by (a) splitting up the implementation artefact, starting with the highest number of $iatan\,(i)$ into several implementation artefacts, referencing to the appropriate feature, by (b) trying to merge features in order to reduce the number of features involved.

There are cases where a resolution is not possible because external components (black box components) or products are used. The main problem here is that the documentation of black box components lacks due to missing information about their interface concerning feature realisation and variability configuration's aspects.

4.4 Variability-Related Implementation Overhead

In real projects several possibilities exist how the design of a system could be structured considering variability. The use of configuration properties in order to be able to set runtime parameters for customers' applications before deployment or build is a simple way to adapt features in their related components e.g. to switch off or on a feature for an application variant. But such a solution often leads to a implementation and configuration overhead which is hard to forecast. Additionally, the resolution of the identified overhead is the more difficult, the more dependencies to features

are constructed. The Implementation Overhead Indicator is able to measure the implementation overhead of such a solution as it measures the elements that are needed to implement an optional feature $f_o \in F$.

Definition: Implementation Overhead Indicator
If no tangling ($tang(a_o) = 0$) and no scattering ($sca(f_o) = 0$) exist the optional feature is build upon elements of exactly one component. This leads to an implementation overhead of 0 (defined through $iov(f_o)$) as such a component could be used as a plugin component obtaining full flexibility concerning variability. In the other cases the implementation overhead is measurable in average using $iov(f_o)$.

$$iov(f_o) = \begin{cases} 0, \text{ if } sca(f_o) = 0 \wedge tang(a_o) = 0 \\ \wedge\ f_o \rightsquigarrow a_o \\ \\ ov(f_o) \in (0, 1], \text{ otherwise.} \end{cases} \tag{19}$$

In this equation the implementation overhead of an optional feature consists of three parts, measuring the involved components, their classes and the used implementation artefacts. The higher the number of artefacts compared to the total number of artefacts is, the higher the implementation overhead (we reuse the definitions $nco(f)$, $nia(f)$ and $ncl(f)$ out of section 3).

$$ov(f_o) = \frac{nco(f_o)}{|3A|} + \frac{nia(f_o)}{|3I|} + \frac{ncl(f_o)}{|3C|}. \tag{20}$$

The more elements are involved in each summand, the higher the implementation overhead gets and the closer the result of the indicator moves to 1. The maximum value 1 is reached if each summand of $ov(f_o)$ is $1/3$, in this case the whole optional feature is implemented through all existing elements and feature tangling exists. An example of the application is given in section 4.6.

Resolution: Reducing implementation overhead

Reducing implementation overhead could be done by (a) splitting up elements (classes and implementation artefacts valued by $ov(f_o)$) of the component implementing tangled features into new separated components or by (b) trying to merge the tangled and optional features to one optional feature in order to resolve the tangling or by (c), trying to merge components that implement parts of the scattered optional feature.

As a consequence the implementation overhead has to be seen in contrast to the ideal case where an 1:1 relation between an optional feature and a component exists and therefore no feature scattering or tangling exists.

4.5 Chain of Components

Because of feature interaction and component dependencies it is possible that a feature could only be used if another feature is also used. In the case of multiple repetitions, a chain of components would occur. However, the more chained components are necessary the higher is the overhead effort for analysis, changes and build. Additionally, the probability of mistakes and errors increases due to changes of features which interact with others.

A small example is shown in figure 5: by adapting a part of the implementation of the component a_3 due to a change of feature f_3 could affect the implementation of f_1 by component a_1, as both components are arranged in the same chain and a class of a_1 uses classes of a_3.

A similar situation could arise during the integration and deployment of implementation artefacts. If products or variants are configured by selecting a set of features, it is important to minimise set of deployed components, in order to decrease the effort of deployment for maintenance and for resource consumption. The Chain Components Indicator shows repeated dependencies and assists their resolution.

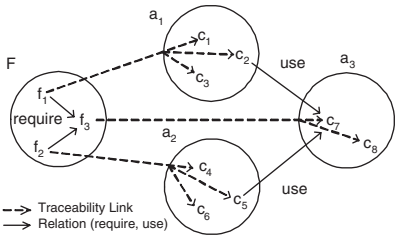


Figure 5. Example of a chain of components

As it is sometimes not possible to reduce dependencies between architectural elements completely, the indicator helps to determine the effort arising from the linked components. The indicator counts for each feature $f \in F$ how many components must be used if selecting f. Formally, this is expressed by the following definitions. Note that in the case of feature scattering of $f \in F$ the components involved by $f \rightsquigarrow a$ are only counted once.

Definition: Transitive Component Chain
The $chain\,(f)$ indicator includes the number of components that are transitively required by the components that implement the feature f. Based on the source component a the equation $rc(a)$ determines the number of directly or indirectly required components $a'' \in A$. This set of components is called Transitive Component Chain and is defined as follows:

$$rc(a) := |\{a'' \in A : \exists a_1, ..., a_n \in A, n \in \mathbb{N}_0 : a \mapsto a_1 \mapsto ... \mapsto a_n \mapsto a''\}|. \tag{21}$$

The indicator $chain(f)$ consists of two parts, the first part is used to determine the number of chained components considering feature scattering by $sca(f)$, the second part is necessary to consider all scattered components $f \rightsquigarrow a$ and sum up the number of chained components by $rc(a)$, respectively. Additionally, $cchain(F)$ is used to analyse all features and to provide an overall measurement result.

Definition: Chain Components Indicator

$$chain\,(f) := sca(f) + \sum_{f \rightsquigarrow a} rc(a), \text{ and} \tag{22}$$

$$cchain(F) := \frac{\sum_{f \in F} chain(f)}{|F| \cdot |A|}, cchain \in [0, 1]\,. \tag{23}$$

If for example f_1 is involved in a product variant (see figure 5) the directly related component has to be used and transitively required components, as well. In the case shown in figure 5 a_3 is transitively related to a_1. Concerning all features in this case $cchain$ is $2/9$ (with $|A| = 3$, $|F| = 3$).

Resolution: Resolution of chained components

Reducing chained components could be done by (a) trying to reduce the number of $rc(a)$ by cutting the trail of components usage, starting with the feature having the highest value of $chain\,(f)$. This could be achieved by dissolving a dependency between two components of the chain. (b) merging components in order to reduce the number of used component for the related feature, (c) creating new features that are implemented by already given components in order to correct or refine the feature structure describing the system architecture more realistic.

4.6 Application of Indicators and Results of the Case Study

In this paper we introduce means for optimizing the design for variability and for measuring the overhead effort. In the previous section, the formal definition of the indicators for several deficiencies were described together with the corresponding resolution actions. These measures have to be applied during the realisation of the system. Since the space limitations inhibit the description of a complete home automation system (HAS), we use a small case study out of such a system to illustrate the application and the benefits of the approach: the possibilities of ensuring consistency between different models, increasing the overall traceability during the system's realisation and evaluating the realisation solution concerning variability and overhead effort.

The case study consists of two cases, *Case A* (see figure 7) and *Case B* (see figure 8), both are used to implement two mandatory features $Modul\ Management$, $Lock\ Closing$ and one optional feature $Control\ Center$ as shown in the

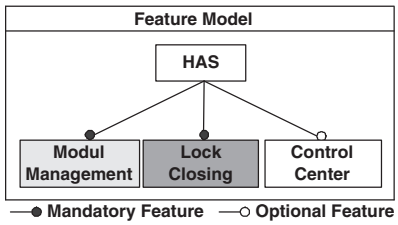


Figure 6. Part of a HAS feature model

feature model in figure 6. Using the defined traceability links in section 3 the realisation of the features is traced in order to be able to map the used realisation elements to its origin. In the figures 6, 7, and 8 each feature and related artefacts are highlighted using the same shade of grey. In the case of feature tangling it is possible that one implementation artefact or class is responsible for different features. As shown in figure 7 the implementation artefact $Configuration$, which is used during the deployment to adapt the feature implementation. It contains the property $viewlimit$ belonging to the feature $Modul\ Management$ and the property $switch$ to disable or enable the optional feature $Control\ Center$ in this component.

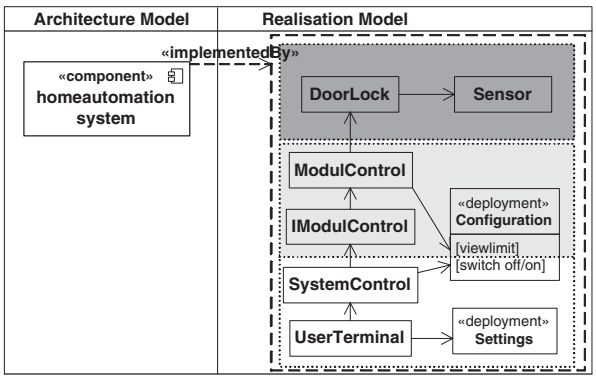


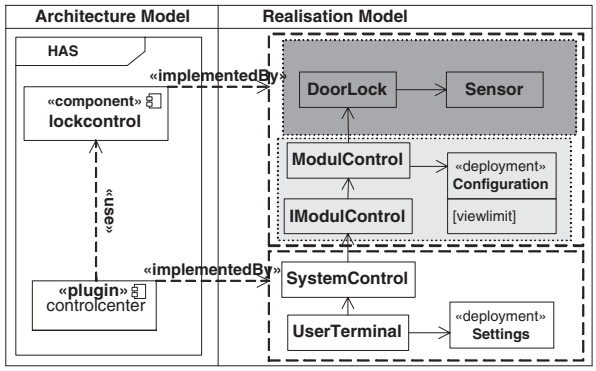
Figure 7. Case A: Constricted variability

The defined indicators are used to analyse this realisation step, the results for the *Case A* are shown in the table presented in figure 7.

First of all feature tangling exists ($ftang = 2/3$) indicating a violation of the concept of separation of concerns. Additionally, the optional feature $Control\ Center$ (in the following abbreviated by f_o) is tangled with the other features resulting to a variability related implementation overhead of about $iov(f_o) = 7/9$. So, it is not possible to plug out the elements related to $Control\ Center$ concerning variability, rather the optional feature $Control\ Center$ is only

switchable by a manual adaption (property switch off/on) of the implementation artefact $Configuration$ (therefore the goal variability is valued as "constricted" in figure 7).

Another disadvantageous situation is given through tangling of the implementation artefacts, valued by $fiatan = 1/6$ in figure 7, as different features use the same implementation artefact. Additionally, valued by the Implementation Artefact Scattering Indicator $fiasca = 1/6$, the adaption of the optional feature $Control\ Center$ have to be done using different configuration units, which increases the probability of misconfiguration during the deployment (as discussed in section 4.3).



Results of Case B:

|A|	|F|	ftang	fiatan	fiasca	iov	cchain	Variability
2	3	1/6	0	0	0	1/6	Flexible

--> Traceability Link
→ Relation

Figure 8. Case B: Flexible variability

The suggested resolution steps are now used to resolve the indicated problem situations, starting with the resolution of the tangled optional feature, by splitting up the related elements into a separated component. This realisation step together with the results of the application of the indicators (*Case B*) are shown in figure 8. As indicated by iov now no variability related implementation overhead exists anymore, gaining full feature flexibility – as it now possible to plug-in/out the component implementing the optional feature. Also, implementation artefact scattering and tangling do not exist anymore: the implementation artefact used by the class $SystemControl$ was only necessary to switch off/on the optional feature, which is not useful anymore. The additional effort for the elimination of feature tangling and variability related implementation overhead leads to a chain (component $controlcenter$ needs $lockcontrol$), which is expressed by the Chain Components Indicator, increasing from 0 to $1/6$. The described re-design makes it possible to use all feature variants without configuration effort. However the achievement of this variability goal leads to a situation of dependency, as the plugin component $controlcenter$ needs the component $lockcontrol$. As a consequence, the component $controlcenter$ is not applicable as standalone component. As in this case the goal variability is the most important requirement, a resolution of the chained components is not useful.

5 Tool Support

The global goals of managing complex tasks with a low overhead effort can only be solved with a efficient support by tools. The evaluations provided by the described method have to be integrated into the design flow. Since software engineering environments often consist of tools from different providers and many of the relevant design tools are available for the Eclipse platform, we decided for this platform. In an earlier project we developed an evaluation tool with a high flexibility of the rules and indicators [13]. The tool accesses the XML model data in a repository. The traceability links are managed as XML data as well. The rules and indicators are expressed either as OCL or as Java expressions, in different versions of the tool. The tool is currently used as an extended prototype. At the current stage of evolution it is flexible, robust and powerful enough to be used in industrial case studies.

6 Related Work

There are some works on artefact relations, especially on traceability relations, for classification purposes, e.g. by Ramesh and Jarke [11]. The problem of a high number of different types of traceability relations has not yet been solved [9]. In this approach we concentrate on a low number types of traceability links according to the goals of their evaluation. The *criteria for flexibility* are part of the design principles for software. From the early software engineering on there have been works in this area, starting from the criteria for modularization by Parnas [8] through the criteria for good object-oriented design [3]. These criteria can be evaluated by a broad variety of design metrics, e.g. collected by Andersson and Vestergren [1] and supported by tools like SDMetrics. But, there is only a limited guidance for using and interpreting these metrics.

By enhancing the flexibility and maintainability of systems, the component-based software development shifts the emphasis from programming code to composing software systems [5]. As an integrated approach with both component technology and domain engineering the software product line techniques have been developed [4, 10] with some approaches covering traceability between features, design artefacts and components [12], similarly to our paper. Within product lines, the variability is modelled explicitly, in many approaches using feature models.

This paper deals with variability, which can be implemented by various typical solutions. Product line technologies distinguish between solutions for different so-called

binding times, the stage of implementation at which a product is instantiated as one choice among several options for a product line [10]. Examples for solutions are already stated in section 2.

There are some methods for managing the mismatch during the integration of (existing) black box components. Guerra et al. [6] propose wrappers covering the mismatch between required and provided behaviour. Other approaches introduce variability both into components and into the component platform, e.g. the Kobra approach [2]. Other component models achieve flexibility by defining a very generalized interface but providing a kind of reflection for information about the actual behaviour of a component, e.g. COM with the IUnknown() interface. In this way they prevent an component interface mismatch by defining a generalized interface, however the mismatch problem still remains for the component behaviour. For a support for the configuration and build of systems with black box components there are approaches for preventing misconfiguration and for testing using interface specifications. The cybersecurity and systems management approach STRIDER [15] for example applies interface specifications called manifests. Currently we do not know about approaches minimizing the configuration overhead with black box components.

7 Conclusions and Future Work

In the way of model-driven design this paper presents an approach for facilitating design for variability by the utilization of model relations. The work is set in the context of software products lines, in which products are mostly built by configuring components. The artefacts and relations relevant for variability, flexibility and product configuration are discussed. Traceability links between several artefacts are investigated concerning their influence on design goals related to variability. Different to many other works in that area, this paper special emphasizes the issues of configuration, build and deployment in addition to issues of architectural design. For the traceability links, a small set of indicators (metrics) has been developed which enables an assessment of models and provides an early feedback during architectural decision-making and reengineering.

The application of these indicators and their integration into a methodical development and configuration process is then briefly illustrated by a small simplified part of a Home Automation System as an industrial case study. The contribution of this paper consists in the support for flexibility and variability by an evaluation-driven design. By reducing the effort for and the impact of changes the evolvability is improved and the long-term value of software systems is increased. The investigation of component relations, configuration issues and the problems of third party components give special support for the development and application of software product lines. As future work we would like to evolve the relations and indicators for different domains. The integration with other works for forwarding constraints between feature model, design, implementation and deployment [12] is planned to improve the integration with configuration tools and to extend the guidance for design and product configuration. The tool integration into decision-making, design, configuration and build processes has to be improved. The latter is planned with an integration of tools with proprietary repositories on the Eclipse platform.

References

[1] M. Andersson and P. Vestergren. Object-oriented design quality metrics. Master's thesis, Uppsala University, 2004.

[2] C. Atkinson and O. Hummel. Towards a methodology for component-driven design. In N. Guelfi, editor, *Proc. RISE Luxembourg, 2004, Revised Selected Papers*, number 3475 in Lecture Notes in Computer Science, pages 23–33, 2005.

[3] G. Booch, R. A. Maksimchuk, and M. W. Engle. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 3rd edition, 2007.

[4] P. Clements and L. M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] P. C. Clements. From subroutines to subsystems: Component-based software development. In *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, pages 3–6. IEEE, 1996.

[6] P. A. de C. Guerra, A. Romanovsky, R. de Lemos, and C. M. F. Rubira. Integrating cots software components into dependable software architectures. In *Proc. ISORC03, Japan*, pages 139–142. IEEE Computer Society Press, 2003.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–58, 1972.

[9] F. A. C. Pinheiro. Requirements traceability. In *Perspectives on Software Requirements*. Springer, 2003.

[10] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering;Foundations, Principles, and Techniques*. Springer-Verlag Berlin Heidelberg, 2005.

[11] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans.*, 27(1):58–93, 2001.

[12] M. Riebisch. Supporting evolutionary development by feature models and traceability links. In *Proc. ECBS2004, Brno, CZ*, pages 370–377, Los Alamitos, USA, 2004.

[13] D. Rohe. Plugin components for development tools (in German: Plugin-Komponenten für Entwicklungswerkzeuge). Master's thesis, TU Ilmenau, 2005.

[14] P. Sochos. *The Feature-Architecture Mapping Method for Feature-Oriented Development of Software Product Lines*. PhD thesis, Technical University of Ilmenau, 2006.

[15] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *Proc. Usenix LISA'2003*, 2003.