

The COVAMOF Software Variability Assessment Method (COSVAM)

Sybren Deelstra, Jos Nijhuis, Jan Bosch, Marco Sinnema
University of Groningen, Department of Mathematics and Computing Science,
P.O. Box 800, 9700 AV The Netherlands
{s.deelstra|j.nijhuis|j.bosch|m.sinnema}@cs.rug.nl

Abstract

Variability has to undergo continual and timely change, or a product family will risk using the ability to effectively exploit the similarities of its members. Being able to determine whether, when and how variability should respond to changing markets, business needs, and advances in technology, however, is a non-trivial task. In this paper, we present COSVAM (The COVAMOF Variability Assessment Method). The contribution of COSVAM is that it provides a technique for variability assessment in the context of evolution, where no techniques were available up to date. It addresses several issues that are associated to the current practice, i.e. ad-hoc and unstructured methodology, implicit variability, addressing only one layer of abstraction, insufficiently exploring alternative solutions, and neglecting implementation dependencies between features.

1. Introduction

The widespread use of product family engineering in industry is governed by the potential increase in quality and productivity of software development in the context of a product family [2][5][9][16]. To achieve this potential, a software product family maintains a product family architecture and set of shared components to exploit the similarities of its members. The product family supports product diversification through variability, i.e. the ability of a software system or artifact to be extended, changed, customized or configured for use in a specific context.

Properly handling this ability, i.e. software variability management, is regarded as a key factor in the success of product families. An important aspect of software variability management is evolution of variability. Over time, variability required from the product family evolves due to, for example, changing markets, changing business needs, and advances in technology. Assessing how the variability in the product family artifacts should respond to this, is a non-trivial problem, however.

In this paper, we present a technique for assessing variability that addresses the issues surrounding the evolution of variability. First, however, we set the scene by explaining variability, and providing a more detailed motivation for the need of a variability assessment technique.

1.1. Variability

Variability is associated to two main aspects, i.e. variation points and dependencies. We introduce both aspects below.

Variation points. Variation points identify locations in software artifacts at which a choice can be made. These identifiers have been recognized as elements that facilitate systematic documentation and traceability of variability, development for and with reuse [3], assessment and evolution. As such, variation points are not by-products of design and implementation of variability, but are identified as central elements in managing variability.

Variation points can occur in all abstraction layers, such as features, architecture, and component implementations. Variation points in one abstraction layer can realize the variability in a higher abstraction level, e.g. an optional architectural component that realizes the choice between two features in the feature tree. Each variation point is associated with zero or more variants and can be categorized to five basic types, i.e. *optional* (zero or one variant out of 1..m associated variants), *alternative* (1 out of 1..m), *optional variant* (0..n out of 1..m), *variant* (1..n out of 1..m), and *value* (a value that can be chosen within a predefined range).

A variation point can be in two different states, i.e. *open* or *closed*. An open variation point is a variation point to which new variants can be added. A closed variation point is a variation to which it is not possible to add new variants. The state of a variation point may change from open to closed in a subsequent phase of the lifecycle. The closing time refers to the latest phase in the lifecycle at which variants can be added.

Variability is either *realized* by variation points in a lower layer of abstraction or implemented by a

realization mechanism. Over the past few years, several of these mechanisms have been identified for different abstraction layers. The realization mechanisms impose a binding time on the associated variation point, i.e. the development phase at which the variation point is bound. Examples realization mechanisms include parameterization, conditional compilation, and dynamically linked libraries (see e.g. [9][15]).

Dependencies. Dependencies are the second main aspect of variability. Dependencies in the context of variability are restrictions on the variant selection of one or more variation points and originate, amongst others, from the application domain (e.g. customer requirements), target platform, implementation details, or restrictions on quality attributes. Although dependencies are often specified as simple ‘exclude’ or ‘requires’ relations, our experience with industrial product families (e.g. [7]) showed that dependencies are often very complex. In many cases, dependencies affect a large number of variation points, and can not be stated formally or exact, but have a less precisely understood character, such as “these combinations of parameters have a negative effect on performance”.

We distinguish between statically analyzable and dynamically analyzable dependencies. Statically analyzable dependencies are dependencies from which the validity can be calculated, and the influence of a reselection of variants can be predicted. Dynamically analyzable dependencies can only be validated by running the system.

1.2. Motivation

As the world around us continually changes, the resulting change in purpose and context may render software products useless. It was therefore that Lehman formulated the following law on software evolution: “*A useful software system must undergo continual and timely change or it risks losing market share*” [11]. This law applies to all products in a product family.

Although variability in the product family architecture and components anticipates some of the changes in space (different products) and time (different versions of products), not all future changes can be predicted or included in the product family. Consequently, once the product family is in place, at some point in the lifecycle, evolution will force the product family to handle new functionality and thus previously discarded or unforeseen differences. In the same way that products need to undergo continual change, variability therefore has to undergo continual and timely change as well, or a product family will risk losing the ability to effectively exploit the similarities of its members.

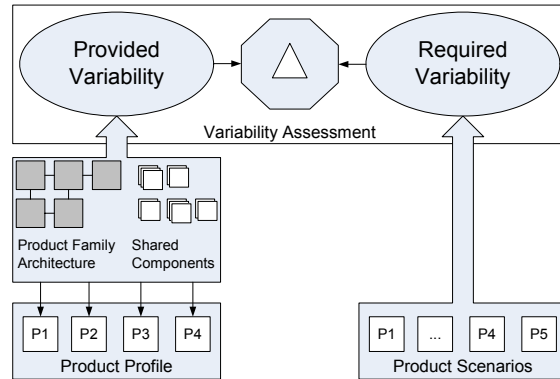


Fig. 1 - Variability assessment. To determine whether, how, and when variability should evolve, variability assessment evaluates the difference between provided and required variability. The assessment can be applied in a number of situations. The situation in this figure involves variability assessment in the context of extending the product portfolio with a new product (P5).

The key challenge in this context is: “how can we determine whether, how, and when variability should evolve?”. A technique that deals with answering this question is what we refer to as variability assessment in the context of an evolving product family. Such a technique answers the question above by analyzing the mismatch between the variability provided by the product family and the variability that is demanded as necessary by the differences in functionality and quality in a set of product scenarios (see Fig. 1).

That variability assessment is indeed relevant becomes clear from examining common activities for software product families in which the ‘whether-how-when’-question appears:

- *Determine the ability of the product family to support a new product:* The decision to add a new product to the portfolio depends on how well the new product fits into the product family scope. This fit depends on to which extent the required combinations of functionality and quality in the new product are supported by the provided variability of the product family.
- *During product derivation, determine whether mismatches should be implemented in product specific artifacts or integrated in the product family:* Variability assessment in this context involves assessing which combinations of functionality and quality of a new product are not supported by the provided variability, and where potential mismatches need to be solved, for example, because mismatches can interact with the variability required by existing product family members.
- *Collecting input data for release planning:* Related to the previous situation is a proactive assessment of provided and required variability

during release planning. Variability assessment in this context involves identifying mismatches as a result of a set of new product releases, as well as determining if mismatches should be solved product specifically or in the product family.

- *Assess the impact of new features that cross-cut the existing product portfolio.* Rather than focusing on adding entire products or product versions, the focus of variability assessment in this context is on assessing the impact of adding one or more features that influence multiple products in the product family.

- *Determine whether all provided variability is still necessary.* In [7], we identified that the existence and lack of removing obsolete variability was one of the underlying causes of complexity, and had a detrimental effect on the efficiency of product derivation. The aim of assessment in this context is to identify provided variability that is obsolete with respect to the variability required by products that have been or will be developed, and to determine how the product family should respond.

Existing literature does recognize the importance of assessing variability [5][16], but suggests using existing techniques, such as change management and architecture assessment, to accomplish this goal. We are the first to provide a technique that is specifically focused on software variability assessment. This technique, COSVAM (COVAMOF [14] Software Variability Assessment Method), solves the shortcomings of existing approaches, such as, ad-hoc and unstructured methodology, focus on only one layer of abstraction, implicit variability, insufficiently exploring alternative solutions, and neglecting implementation dependencies between features. The COSVAM consists of five main steps, i.e. set assessment goal, specify provided variability, specify required variability, evaluation of mismatches between provided and required variability and, finally, interpretation of assessment results.

1.3. Remainder of this paper

In the following section, we present a number of issues that are faced when assessing variability. In section 3, we discuss related work. In section 4, we briefly present a case study at Dacolian B.V. In section 5, we present COSVAM. In section 6, we illustrate our assessment technique with the case study at Dacolian B.V. We conclude by discussing how COSVAM addresses the issues identified in section 3.

2. Problem statement

Assessing variability in the context of evolution is a non-trivial task. We identify a number of issues that are associated to the current practice.

Ad-hoc and unstructured methodology. As product families in industry have been widely adopted and evolve constantly, most organizations that employ product families already perform some form of variability assessment. Before a new product is derived, for example, a specification of the product features is handed to, typically, software architects. The task of these architects is to assess how much effort will be associated in delivering the product with this specific set of features. Such an assessment is often done by these architects without explicit methodological or process guidance. The drawback of ad-hoc assessments is that they often lead to product specific adaptations that neglect all the benefits of a software product family. In addition, changes may lead to incompatibility with the asset base of the software product family. The result is that often during product derivation, unexpected incompatibilities are identified, which have a profound impact on the total effort and time-to-market for the product at hand [7].

Addressing only one layer of abstraction. Most existing assessment techniques focus on only one layer of abstraction, i.e. either the architecture, or the source code level. As variability is a crosscutting concern, however, there is a need for a technique that uniformly treats variability at all layers of abstraction.

Implicit variability. In many organizations, no complete and explicit variability model is available that covers all phases of the lifecycle. As time and effort available for an assessment is limited, specifying a complete explicit model is often not an option.

Insufficiently exploring alternative solutions. When a variability mismatch occurs, several solution strategies may exist. An architect can, for example, choose from different realization mechanisms to add a new variation point, and decide whether to solve a mismatch product specifically or in the reuse infrastructure. The choice for a particular solution depends on a trade-off between pros and cons of the potential solutions, as certain solutions may, for example, introduce incompatibilities in the asset base due to new dependencies, and affect the effort associated with adding other features.

Neglecting implementation dependencies between features. Even if features are independent from a problem space perspective, the design and implementation of a product family creates additional interactions between features. The consequence of dependencies as a result of implementation is twofold. First, not all combinations of the features provided by a product family can be offered in one product without modification. Second, effort estimates for new features cannot be considered independent from other changes.

3. Related Work

A number of approaches are related to variability assessment. In this section, we discuss the shortcomings of a number of these approaches.

FAST and SEI's Product Line Practice. In [16], Weiss and Lai formulate basic assumptions with respect to product families, from which two are particularly important in the context of this article: "it is possible to predict the changes that are likely to be needed to a system over its lifecycle", and "it is possible to take advantage of these predicted changes". When it comes to actually determining changes to variability, however, the book lacks precision (p. 198): "... You can adapt standard change management techniques to FAST projects, so the FAST PASTA model does not elaborate on those aspects in any great detail" [16]. Also in the SEI's Product Line Practices and Patterns book, the evaluation of variability is quoted as an example of an important evaluation. The book, does not present a technique to perform these evaluations. Rather, it suggests modifying existing architecture assessment methods to accomplish this goal (pp. 77-83).

Investment analysis. In [17], James Whitey provides an investment analysis approach that focuses on maximizing ROI of product line assets. Robertson and Ulrich [13] also evaluate economical aspects of a product family and deal with planning and scoping the product family architecture. In [6], however, DeBaud and Schmid provide a similar but more general approach. They also claim that product-centric commonality and variability analysis is better than a domain based view, as the latter provides a flawed economic model for making scoping decisions [6]. Each of the approaches, however, is based on rough estimates for implementing features that only consider product specific vs. product family as possible solutions, and assume that the effort for implementing a feature is independent of how other features are implemented.

Assessment. Assessments typically consist of five steps, i.e. goal specification, specification of the provided aspect, specification of the required aspect,

analyzing the difference between the provided and required aspect, and interpreting the results. Examples of these approaches are ATAM [4], ALMA [1], and SALUTA [8]. These three approaches respectively assess trade-offs between quality attributes, maintainability, and usability in software architectures. The approaches differ with respect to the required information, elicitation and specification of the scenarios. They focus on analyzing the architecture of single systems.

4. Case study

The COSVAM was applied at Dacolian B.V. Dacolian is a Small-To-Medium (SME) in the Netherlands that mainly develops intellectual property (IP)-software modules for intelligent traffic systems. The focus for the assessment was on the Intrada® product family whose members use outdoor video or still images as most important sensor input (see <http://www.dacolian.nl>).

Dacolian maintains a product family architecture and in-house developed reusable components that are used for product construction. The infrastructure contains special designed tooling for Model Driven Architecture (MDA)-based code generation, software for module calibration, dedicated scripts and tooling for product, feature and component testing, and large data repositories. These assets capture functionality common to either all or a subset of Intrada products.

Binding is typically done by configuration and license files at start-up time, as well as MDA code generation, pre-compiler directives and Make dependencies for pre-deployment phases.

5. COSVAM: The COVAMOF Software Variability Assessment Method

To address the aforementioned issues, we developed the COVAMOF Software Variability Assessment Method (COSVAM). COSVAM adopts the standard steps of scenario-based assessment techniques and draws from the experience of existing assessment methods. COSVAM is an iterative method, consisting of 5 steps (see also Fig. 2).

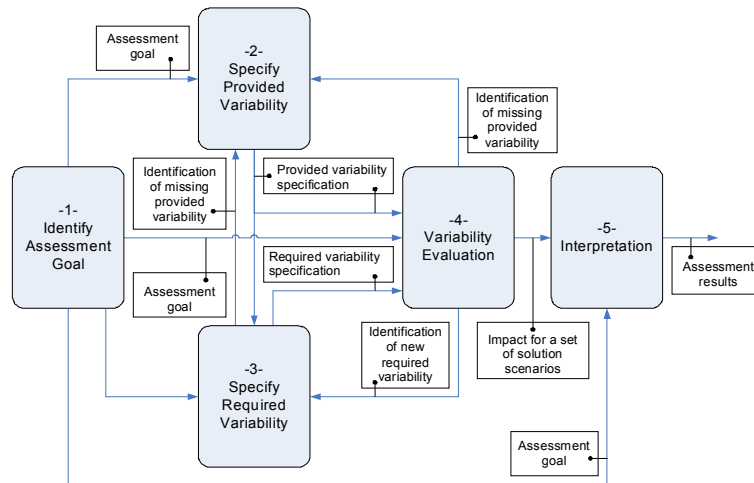


Fig. 2 The structure of COSVAM. COSVAM is an iterative method that consists of 5 steps. The arrows in this figure denote the main results and iterations between steps.

5.1. Identify assessment goal.

Software variability assessment is used in different contexts, each with a different goal. The goal has an impact on all assessment steps.

Initiation: An assessment is typically initiated for three reasons. (1) A *periodic* assessment is regularly initiated, e.g. once per month, or once per year. Examples include assessment for release planning, and benchmarking the organization. (2) An *event-driven* assessment is marked by a significant event. Examples include the start of deriving a new product version, and extending the product portfolio. (3) A *stakeholder-initiated* assessment is initiated by one or more influential members of the organization. An example is a reverse assessment after complaints about the high complexity due to obsolete variability.

Influence: The initiation can have a considerable impact on the required input and cooperation during the assessment, time pressure, and acceptance of the results afterwards. In a periodic assessment, for example, the organization is typically committed to the assessment, while in an event-driven or stakeholder-initiated assessment not all members of the organization may see the need, and the process potentially suffers from time pressure, political forces, and lack of commitment.

Define assessment outcome: The second activity of identifying the assessment goal is identifying the type of results that are expected from the assessment, such as, a list of specific mismatches between provided and required variability, an overview of possible solutions for these variability mismatches, effort estimates, or a qualitative measurement. The desired assessment outcome influences the extent in which the COSVAM process is executed. When a list of

mismatches is required, for example, the fourth step (analyze mismatches) can skip the activities in which different solutions are considered, and effort is estimated.

Select scope: A complete assessment involves the entire product family, all products, product versions and their features, the entire product family architecture, and all shared components. As the time and effort for the assessment is limited, however, the assessment can be limited by scoping the provided and required variability.

The *required variability scope* is determined by the size of the product scenario set:

- *A product portfolio subset:* Instead of using all products for the assessment, one product or a subset can be chosen to serve as input for the required variability. The assessment is typically limited to one product if the purpose of the assessment is to identify how well that product is supported by the product family, and which product specific changes should be made to accommodate mismatches. For identifying product family changes, a subset of products can be used. The subset can be focused on future products that will use the next release of the product family, future products that span across different releases, or only present-day products. In addition, old products can be used to identify whether all variability was really necessary.
- *A subset of features:* The level of detail of the required variability can be limited further by focusing on the key product features.

The *provided variability scope* is determined by the parts of the product family that are considered during the assessment. The result is that both the amount of information that has to be externalized and managed considerably decreases, and that fewer experts have to be involved in the assessment.

- A *horizontal slice* limits the assessment to the product family architecture. Many implementation details can be left implicit, and the assessment primarily depends on the involvement of architects.
- A *vertical slice* limits the assessment to a particular subsystem of the product family, thus requiring only a portion of the component experts or application engineers in externalizing the provided variability.

Note that while limiting the provided and required variability scope limits the time, effort, and number of key players that should be involved in the assessment, it does restrict the level of detail of the assessment and thus the associated accuracy of results.

Identify members of the assessment team: Based on the purpose and scope of the assessment, the assessment team has to be extended with a variety of experts. These experts are needed, as the assessment involves a situation where much of the information is implicit, and deals with predicting the future and estimating effort. Examples of experts thus include software architects to identify architectural variability, product developers that know existing product configurations, and have experience with the provided variability of the product family, component experts to externalize component variability, technology experts and marketing managers to provide predictions with respect to future products, and project managers to provide productivity figures.

The combined results from these steps form the goal of the assessment. Once this goal has been specified, it is important to plan the assessment, obtain commitment from the organization, and to assemble the assessment team. For a more detailed discussion on these aspects see e.g. [4][12].

5.2. Specify provided variability

The purpose of the second step in COSVAM is to specify the provided variability that is relevant for the assessment, in a form where variation points and dependencies are first-class and related across lifecycle phases. In case a provided variability model is already available in the right form and detail, the assessment can proceed with specifying required variability.

Throughout this step, the assessment team continually has to weigh how much information will be externalized and how much will be left implicit. The team documents any decisions to partially externalize certain variation points or complete parts of the product family. When during a later stage in COSVAM it turns out that information contained in these parts is necessary, the assessment team can decide to iterate this specification step (see Fig 2), or

leave the information implicit. The decisions to leave parts implicit are weighed in the interpretation step of the assessment.

The provided variability specification process is structured as follows: *identify and obtain information sources, identify variation points, unify set of variation points, identify variants, identify dependencies, collect remaining properties.*

Identify and obtain information sources: The specification step starts with identifying information sources that can be used to externalize the, for a large part, implicit variability. Potential information sources include: expert knowledge, feature diagrams, the product family architecture, component documentation, product configurations, make-files, and source code.

Identify variation points: Once the information sources are obtained, the next activity is to identify the first of the two aspects of variability, i.e. the variation points. A number of approaches can be used:

- Some existing notations, such as for feature diagrams, already specify variability. Variation points can be identified directly from these notations.
- Interview experts, and let them draw from experience to identify important variation points.
- Compare the architecture, selected components, and parameter settings in different product configurations to identify differences.
- Search for ‘patterns’ in design and implementation that may indicate variability. These patterns consist of design patterns such as factories and abstract classes, but also code statements such as if-statements and pre-compiler directives
- Search for comments that deal with choices in design and code. Although specifying criteria for an automated search with full-coverage may prove complicated, comments can be identified by experts, and often serve as valuable addition to the other means for externalizing variability above.

Unify set of variation points: The identification of variation points results in a set of variation points that originate from different sources. The purpose of unification is to identify multiple representations of the same variation point, to relate variation points across levels, to remove too low-level variation points, to identify the type, when the variation points are bound, whether they is open or closed, and what realization technique is used. The unification process furthermore identifies settings that are identical across multiple product configurations. These variation points are marked as potentially obsolete.

Identify variants: The unified set of variation points is then populated with variants. A number of variants

have already been encountered earlier, as these variants served to identify variation points in the first place. This set of variants is not complete however, as only a subset of product configurations may have been used, or because the product configurations only use a subset of the available variants. The set of variants can therefore be extended by interviewing experts, and by inspecting additional product configurations and the set of shared product family assets. This process is guided by the identified variation points.

Identify dependencies: Variants and variation points are related across levels. Different combinations of variants at architecture and component level realize different combinations of features, and result in different quality attributes. In addition, not all combinations of variants permitted due to dependencies or exclusions. These realization relations and other dependencies can be determined by interviewing experts, static and dynamic source code analysis, studying comments in, for example, configuration files, and inspecting product configurations in order to determine whether certain combinations of variants always change in pairs.

5.3. Specify required variability

The third step in COSVAM is the specification of required variability. This specification captures the variability that is required to accommodate the combinations of functionality and quality in the set of product family members that is in the scope of the assessment. The process for specifying required variability consists of *identifying the information sources, construction of product scenarios, and constructing the specification.*

Identify and obtain information sources: The specification process starts with selecting the information sources that are necessary to determine the required variability. The selection of these sources depends on the required variability scope. Example sources include existing and future product specifications, market analysis, different roadmaps, and a catalog of requirements requested by customers.

Construct product scenarios: The next activity is to construct the product scenarios. Product scenarios consist of an overview of the features of products that are within the scope of the assessment. The construction process consists of:

- Identifying a set of required features, a set of quality attributes and identifying a set of required products. To ensure that the set of features and quality attributes have a consistent meaning in the provided and required specification, a feature

dictionary is maintained (see also domain analysis methods such as FODA [10]).

- For each product, selecting a combination features that is likely to be needed for the product and that is relevant for the scope of the assessment.
- Identifying variation within each product: the type of variation, and the earliest and latest time each variant should be bound (e.g. compile-time, runtime or don't care).
- Identifying required quality for each product.

Depending on how much information is already available, one or more results of this construction process may be part of the input of this step.

Construct specification: The product scenarios and provided variability specification are used to construct the required variability specification.

- *Select features:* For all features or set of alternatives in each product scenario, select the feature or the set of alternatives.
- *Identify new variation points:* If the features will not be part of an existing variation point in the provided or required variability specification they are part of a new variation point.
- *Identify new variants:* If the features are not already part of the provided or required variability specification, it is a new required variant.
- *Mark variants:* Each variant that is required by a product is marked with the product identifier. These variants can include variants at variation points that are already provided by the product family.
- *Specify variation point attributes:* Based on the variability within the product scenarios, specify the earliest & latest required binding time, and required type per variation point per product.
- *Specify required quality:* The important quality attributes are specified with dependencies. For each product, the corresponding dependency is complemented with the required value, range, maximum or minimum quality.

Note that the result of this step is a specification of required variability in terms of a *delta* to the provided variability: the required variability is specified in terms of provided variants that need to be bound in the individual product family members, plus a delta to the provided variability in terms of new variation points and changes to the attributes of existing variation points.

5.4. Evaluate required and provided variability

The fourth step in the assessment is to evaluate the provided and required variability. The purpose of this step is to find out how well the required variability is supported by the provided variability, and what changes should be made to accommodate mismatches

between them. The evaluation process is structured as follows: *identify direct mismatches, identify indirect mismatches, cluster and prioritize related mismatches, devise a set of possible solutions, and determine the impact of possible solutions.*

Identify direct mismatches: New variation points and variants are mismatches that can be identified directly from the required variability specification.

Identify indirect mismatches: Other mismatches require comparing the provided and required variability. A number of actions are required:

- The realization relations between variation points are traced to identify how choices at the feature level translate to choices at architecture and component levels. As a result, required variants are identified at lower levels. These are added to the required variability specification.
- Inspecting dependencies to determine whether conflicts occur between variants, or whether required quality is met after propagating the choices.
- Inspecting whether the provided binding time is within the windows of earliest and latest binding time to identify binding time mismatches.
- Inspecting whether the provided attributes match the required attributes to identify mismatches in type and variant addition time.

Cluster and prioritize mismatches: The result of the mismatch identification step is a list of mismatches. Often, the solutions to these mismatches are related, for example, when the mismatches involve existing variation points that are dependent on each other, or when the mismatches involve variation points in the same components. The relations between these mismatches often drive the necessity for iterations during impact analysis. To minimize the number of iterations in the analysis, the mismatches are clustered into relatively independent impact analysis sets.

The relations between mismatches are found by inspecting the provided variability to determine where variation points are located, and which the dependencies exist between them. After clustering, the impact analysis sets are prioritized according to the likelihood that the results of the analysis process will affect the analysis of other sets. This likelihood is based on aspects such as the type of mismatches, technical difficulty of the involved variability, whether they involve system-wide quality attributes, or the number of dependencies.

Note that clustering and prioritization continues during the analysis, where new relations are discovered and created between variation points. Impact analysis sets may furthermore be combined or split if during the analysis it turns out they are

respectively more or less related than the assessment team initially expected.

Devise a set of possible solutions for each impact analysis set: The next activity involves identifying changes that are necessary to accommodate the mismatches in the impact analysis set. Frequently, several solutions exist for accommodating these variability mismatches. Important differences in solutions are, for example, the decision to solve all mismatches product specifically, or to solve all or a part of the mismatches in the product family. Other important decisions are the binding time and realization mechanism that are chosen. The purpose of this step is therefore to identify affected entities under different realization scenarios, and to determine the necessary changes on these entities.

This step is highly dependent on expert knowledge and creativity of architects and designers. The process can be supported by several techniques, however. For variant mismatches at existing variation points or binding time mismatches, for example, the provided variability specification can be used to trace realization relations to identify the locations where the variation point is implemented, as well as which realization mechanisms are used. Source code analysis on the other hand can be used to determine affected entities when replacing stable with variability components, or changing variability realization mechanisms.

Note that the different solutions for mismatches at a higher level may require new variability at lower levels. This causes iterations between the required variability specification step and the evaluation (see Fig. 2).

Determine the impact of possible solutions: Each of the realization scenarios identified above has a different impact. They differ in terms of effort associated with product derivation and product family development, have a local or global effect on the design, influence the time-to-market of products, and result in different quality attributes. Products may furthermore impose conflicting requirements with respect to variability, and different solutions solve conflicts differently. Techniques used to determine these impacts depend on the desired assessment outcome and vary across organizations. COSVAM therefore does not prescribe a particular technique.

5.5. Interpret the evaluation results

The purpose of the interpretation is to draw conclusions based on the evaluation. We limit the discussion in this paper to the situation in which the optimal solution scenarios need to be selected. The final step of COSVAM consists of *identifying*

relevant business drivers and constraints, identifying pros and cons of different solutions, and selecting a solution.

Identify relevant business goals and constraints:

The interpretation starts with identifying aspects that have an influence on the conclusion. For identifying the optimal solution scenarios these are constraints such as size of development teams, the maturity of involved technology, schedules, and business goals such as minimizing the number of times products are compiled, reduction of time-to-market, and ease of maintenance.

Identify the pros and cons of the different solutions:

The next activity is to get a list of pros and cons for each solution scenario, by documenting the achievements and remaining conflicts, considering how much information was left implicit in identifying the mismatches and solution, as well as identifying how the impact of different solution scenarios relates to the business goals and constraints, such as the effect on the development schedules, risk of failure, liability issues, fault detection, and time-to-market.

Select solutions: The pros and cons of the different solutions identified in previous steps are then weighed, and the solutions that optimize the constraints and business goals are selected.

6. Validation and illustration

Dacolian applied COSVAM on their Intrada product family (see also section 4). In this section, we illustrate the various steps of COSVAM using this case. For reasons of space, we focus on a key variation point in Dacolian's Intrada product family: the variation point regarding the recognition of license plates of various countries.

Identify assessment goal: The variability assessment was initiated to plan the new release of the Intrada product family. The new release should be due in the next 4 months, and the design and development time was the major constraint. The provided variability scope excluded the platform layers, and the required variability scope consisted of both new and existing products and product versions. The outcome of the assessment should be a selection of the best solution.

Specify provided variability: The main sources for the provided variability specification were c-interface definitions, product documentation, and experts. The specification process identified that the Intrada product family contained approximately 7000 variation points, and that many were used to

configure the countries variation point. The alternative countries variation point at the feature level had a closing time at compile time, and supported 5 existing variants.

Specify required variability: The product scenarios consisted of more than 100 product and product version specifications that required different combinations of countries to be recognized. This resulted in 31 new variants to be required at the country variation point, and a change of attribute from alternative to multiple coexisting variants.

Evaluate required and provided variability: The evaluation step revealed that with the current provided variability, for each new country, and each different combination of countries that was required, a new compiled system had to be built. A first proposed solution to this problem was to leave things as they were, and to automate the configuration process to accommodate for many different product versions. A quick calculation, however, showed that realizing the automation would be easy, but that the number of product variations would explode very rapidly, and product maintenance would become rather impossible.

To address this problem, it was identified that countries should become a variation point that had a closing time at startup time instead, and the primary mechanism used to realize the involved variation points should be based on dynamically linked libraries. Several required products had stricter memory requirements than the others, however, which drove the need for a new variation point that would allow for a product version that does not need dynamic libraries.

Due to the complex and many realization relations that are associated with the countries variation point this meant that the closing time of many other variation points should be at startup time as well. In total, 231 variation points were involved that needed to have their closing time changed from compile time to startup time.

Two other solutions have thus been considered that use the dynamic libraries. The second solution was to place routines that are functionally equivalent in separate dynamic libraries, while the third was to group all functionality related to a single country into a separate dynamic library.

Solution 2 fits well within the current architecture of the Intrada PF, as routines with similar functionality are grouped together. However, also of this solution, adding a new country would require changes to parts of the software that could have been untouched as they have no direct link with the new country.

Solution 3, on the other hand, would require a global change to the architecture, as routines would

now be grouped as countries. A major risk of this change was the potential lack of reuse of already calculated results during execution, which would result in a slower system. The major advantage of this solution would be that it is possible to develop new country variants independent of the already existing code.

Interpretation: The assessment team estimated that the realization and testing of each of the three solutions would cost several months and would be the same for each of the solutions. The risks with respect to meeting quality requirements, however, would be minimal for solution 1, and the highest for solution 3. The major benefit of solution 3 over the other two solutions was that the development effort and time-to-market for products requiring new country variants would significantly decrease. Due to a drastic increase in the number of request for new countries, the decrease in time-to-market, and possibility for independent development of country variants, outweighed the risk associated with solution 3. Solution 3 was therefore chosen to address the required variability during the lifecycle of the new release.

7. Discussion and conclusion

The predominant challenge, in most software product families, is the management of the variability required to facilitate the product differences. Over time, the variability required from the product family evolves, but assessing how the variability provided by the product family artifacts needs to evolve in response, is a non-trivial problem. Currently, however, there is a lack of techniques that specifically support the assessment of software variability in product family artifacts. In response to this, we have developed COSVAM, the COVAMOF Software Variability Assessment Method.

7.1. Contribution

The contribution of COSVAM is that it is the first technique for assessing variability. It addresses the issues we presented in section 3 as follows:

- *Ad-hoc and unstructured practice:* COSVAM addresses this issue by distinguishing multiple goals, defining repeatable steps, and forcing explicit assumptions and decisions.
- *Addressing only one layer of abstraction:* The COSVAM is built around a notion of variability that treats variation points and dependencies uniformly as first-class citizens that are related across abstraction layers. The COSVAM is aimed at all abstraction layers, but allows for focusing on a subset by selecting the appropriate assessment scope.

- *Implicit variability:* COSVAM provides an iterative process that is focused on minimizing effort with respect to externalizing variability that is irrelevant for the assessment. COSVAM explicitly deals with missing information by documenting which parts are deliberately left implicit and weighing missing information in the interpretation step.

- *Insufficiently exploring alternative solutions:* The COSVAM explicitly deals with this issue by devising multiple solution strategies and weighing pros and cons of different realization scenarios.

- *Neglecting implementation dependencies between features:* During the evaluation, it is verified whether the dependencies in the provided variability allow for offering the required combinations, and, in case of mismatches, how different solutions address these implementation dependencies.

7.2. Future work

So far, we have briefly discussed the inaccuracy of provided variability. In case the assessment involves future products, however, the required variability is also subject to inaccuracy. Predicted features may not be needed, for example, or new products may require a different set of features and quality than predicted. We are currently extending the COSVAM to deal with inaccuracy of predictions, as well as ways to quantify the inaccuracy of provided and required variability. In addition, we intend to investigate existing quantitative and qualitative models for weighing the pros and cons of different solutions scenarios. Finally, although the technique has been applied on the case provided by Dacolian B.V., we will apply it on several additional cases in the future.

Acknowledgements: This research has been sponsored by ConIPF, contract no. IST-2001-34438.

8. References

- [1] Bengtsson, P.O., Lassing, N., Bosch, J., van Vliet, H., "Architecture-level Modifiability Analysis (ALMA)", *Journal of Systems and Software*, Vol. 69(1-2), pp. 129-147, Jan 2004.
- [2] Bosch, J., *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, 2000.
- [3] Clauss, M., "Modeling variability with UML", GCSE 2001 - Young Researchers Workshop, Sept. 2001.
- [4] Clements, P., Kazman, R., Klein, M., *Evaluating Software Architectures, Methods and Case Studies*, Addison-Wesley, ISBN 0-201-70482-X, 2001.
- [5] Clements, P., Northrop, L., *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7, 2001.
- [6] DeBaud, J.M., Schmid, K., "A systematic approach to derive the scope of software product lines", *Proceedings of*

- the 21st Int. Conf. on Software Engineering, California, USA, May 1999, pp. 34-43.
- [7] Deelstra, S., Sinnema, M., Bosch, J., "Product Derivation in Software Product Families, A Case Study", accepted for the Journal of Systems and Software, 2003.
- [8] Folmer, E., Gulp, J., Bosch, J., "Architecture-Level Usability Assessment", accepted for EHCI, July 2004.
- [9] Jacobson, I., Griss, M., Jonsson, P., Software Reuse. Architecture, Process and Organization for Business Success, Addison-Wesley, ISBN: 0-201-92476-5, 1997.
- [10] Kang K., Cohen S., Hess J., Nowak W., and Peterson S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, SEI, 1990.
- [11] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M., "Metrics and Laws of Software Evolution - The Nineties View", Proceedings of the Fourth International Software Metrics Symposium, USA, Nov. 1997.
- [12] Maccari, A., "Experiences in assessing product family software architecture for evolution", Proceedings of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, pp. 585-592, 2002.
- [13] Robertson, D. Ulrich, K., "Planning for product platforms", Sloan Mgt. Review, Vol. 39 (4), 1998, pp. 19-31.
- [14] Sinnema, M., Deelstra, S., Nijhuis, J.A.G., Bosch, J., "COVAMOF: A Framework for Modeling Variability in Software Product Families", accepted for the 3rd Software Product Line Conference, Boston, USA, 2004.
- [15] Svahnberg, M., Gulp, J. van, Bosch, J., "A Taxonomy of Variability Realization Techniques", ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.
- [16] Weiss, D.M., Lai, C.T.R., Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley, ISBN 0-201-694387, 1999.
- [17] Whitey, J., "Investment analysis of software assets for product lines", Technical report CMU/SEI-96-TR-010, Software Engineering Institute, 1996.