Modeling Dependencies in Product Families with COVAMOF

Marco Sinnema, Sybren Deelstra, Jos Nijhuis
University of Groningen
P.O. Box 800, 9700 AV,
The Netherlands
{m.sinnema|s.k.deelstra|j.a.g.nijhuis}@rug.nl

Jan Bosch
Nokia Research Center
P.O. Box 407, FI-00045 NOKIA GROUP,
Finland
jan.bosch@nokia.com

Abstract

Many variability modeling approaches consider only formalized dependencies, i.e. in- or exclude relations between variants. However, in real industrial product families, dependencies are often much more complicated. In this paper, we discuss the product derivation problems associated with dependencies, and show how our variability modeling framework COVAMOF addresses these issues. Throughout the paper, we use examples of Intrada, an intelligent traffic systems family of Dacolian B.V.

1. Introduction

Variability is the ability of a software system or artifact to be extended, changed, customized, or configured for use in a specific context. Variability plays a key role in the success of a software product family, as it enables deriving different products. On the one hand, variability is enabled through variation points, i.e. the locations in the software that enable choice at different abstraction layers. Each variation point is associated with a number of options to choose from (called variants). On the other hand, the possible configurations are restricted due to dependencies that exist between variants, and the constraints that are imposed upon these dependencies.

The implicit or unknown properties, and the almost unmanageable amount of variability in terms of sheer numbers (ranging up to ten thousands of variation points and dependencies), are two core issues that have a detrimental effect on the time, effort, and cost associated with product derivation. In addition, the implicit and unknown properties make the product derivation process highly dependent on experts [7].

A large part of this problem is associated with dependencies. Most existing variability modeling approaches (e.g. [1][2][3][4][10][14][18]) only address the formalization of dependencies. Formalization, however, does not address the most challenging problems in variability management, viz. challenges that result from dealing with imprecise and incomplete knowledge.

To deal with these issues, we developed our modeling framework COVAMOF. In earlier work (e.g. [14]), we primarily focused on modeling variability in general, and variation points in particular. The contribution of this paper is that we provide a detailed discussion on the relation between dependencies and challenges that are faced during product derivation. We furthermore show how modeling of dependencies with COVAMOF addresses these challenges. To this purpose, this paper is organized as follows. In section 2, we describe the problem in terms of types of knowledge that exist for dependencies, and discuss the influence of these types and availability of knowledge on the product derivation process. In section 3, we discuss how we address these problems in our variability modeling framework COVAMOF. In section 4, we discuss the benefits of the solution offered by COVAMOF. We conclude our paper in section 5.

2. Dependencies and Product Derivation

Product derivation involves, amongst others, selecting variants at the variation points in a software product family. These variants are selected in such a way that the resulting product satisfies the product requirements. In [7] and [8], we described how products are derived in software product families (see also Figure 1). Typically, a product derivation process starts with an initial phase, where a first configuration of the product is derived using assembly (construction, or generation), configuration selection, or any combination of both. Although the initial configuration usually provides a sufficient basis for continuing the process, the initial configuration is often not finished.

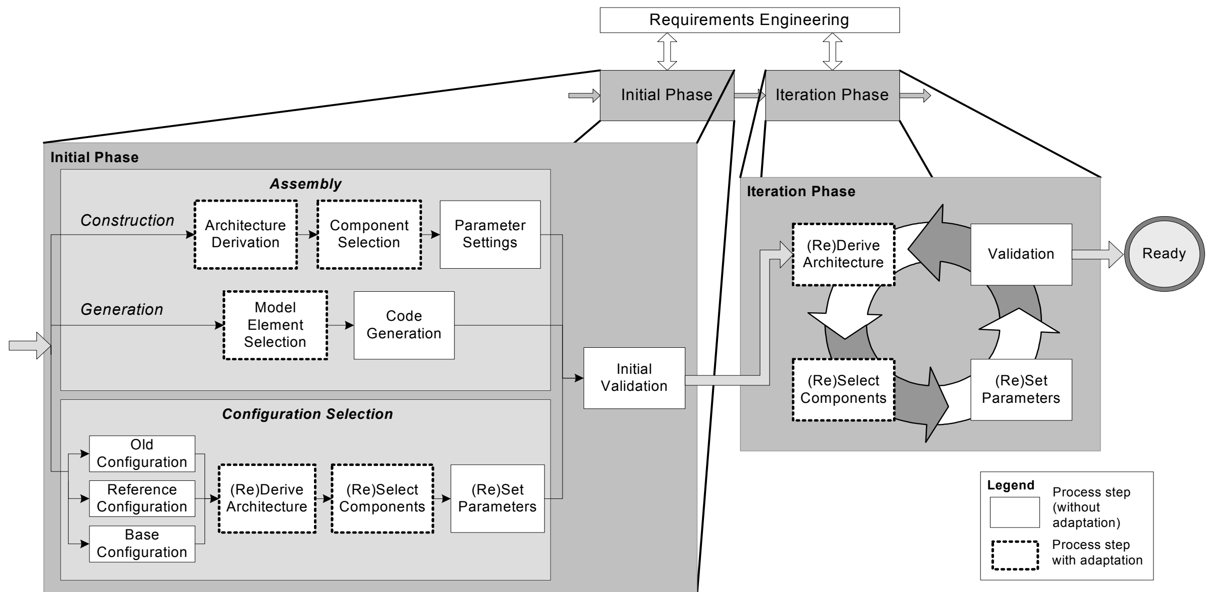

Figure 1. Product derivation in software product families. A typical product derivation process starts with an initial phase, where a first configuration is constructed using assembly or configuration selection. The process usually goes through a number of iterations before the product is finished.

The process then enters the iteration phase, where the configuration is changed until the product is ready.

The role of dependencies in this context is that dependencies are relations between a set of variation points. The dependencies specify a system property whose value is based on the selection of variants at the different variation points. A dependency can, for example, specify a property that tells whether a particular set of selected variants is compatible with each other, or a property that specifies the value of a quality attribute such as performance or memory usage.

During product derivation, engineers specify constraints on the value of these properties, such as "the memory usage should be smaller than 64 MB", or "the compatibility of variants should be TRUE". These constraints originate from product requirements. As multiple dependencies can influence a particular choice, often not all constraints on the properties can immediately be met for a particular product (if they can at all). In the following subsections, we discuss the consequence of this fact with respect to different types and availability of knowledge.

2.1. Knowledge types: formalized, documented and tacit knowledge

We start our discussion on knowledge types and dependencies with an example of the Intrada product family. The Intrada product family is a family of intellectual property software modules for intelligent traffic systems that is developed by Dacolian B.V. The reusable asset base of this family consists of approximately 11 million lines of code. Typical products consists of systems that deliver, based on real-time input images, abstract information on the actual contents of the images, e.g. for the detection of moving traffic, vehicle type classification, license plate reading, video based tolling, and parking.

Example 1: *Several parts of the configuration of an Intrada product are partly or completely automated. This automation is based on the extensive use of formalized knowledge. Examples of formalized knowledge exist at all stages of the configuration process:*

- *At pre-compile time, #ifdef and #define preprocessor directives are used to make sure that the platform and operating system dependencies are correctly handled. This means that the appropriate include files, sources, and constants are supplied automatically when the designer selects a platform/operating system combination.*
- *For devices with a limited amount of memory, e.g. PDA's, Intrada products require different versions of modules to be linked. Specialized tooling is used to generate the correct Makefiles.*

The knowledge that is available for the dependencies in the example above is an illustration of what we call formal knowledge, i.e. explicit knowledge that is written down in a formal language, and that can be used and interpreted by computer applications. Existing variability modeling approaches are only based on this formalized knowledge [15].

In product families, however, two other types of knowledge exists as well, i.e. tacit knowledge, and documented knowledge. Tacit knowledge [12] is information that exists in the minds of the engineers that are involved in product derivation, but that is not written down in any form. Documented knowledge is information that is expressed in informal models and descriptions. To exemplify both types, we provide two examples below.

Example 2: *The derivation of high performance Intrada products as being used in large tolling projects is a complicated task. Engineers have established that there are indeed dependencies between the various configuration options, and that tradeoffs have to be made. Graphs like the one in Figure 2 are created for various configurations to score the actual configuration for the dependencies. Dacolian has build tooling to assess a configuration (Intrada Performance Analyser) but only a few experts are capable of performing a directed optimization during the configuration process, as this knowledge is currently only available as tacit knowledge. Where product derivation for a typical product only takes a few hours at maximum, these complicated product derivations take up to several months.*

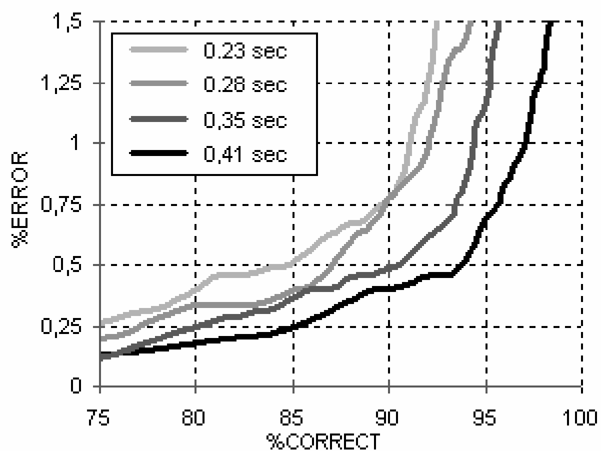

Figure 2. The effect of varying maximum processing time on a typical configuration. This is an example of partially tacit, and partially documented knowledge. Based on such graphs, less experienced application engineers are able to predict the influence of the effect of varying the maximum processing time for recognizing an image. As soon as the maximum processing time becomes a real issue (e.g. a maximum of 0.23 seconds with a correct rate of 95%), only experts know which actions to take in order to perform a directed optimization towards the desired results.

Example 3: *Some of the other complex dependencies at Dacolian have also been documented so that less experienced engineers are also capable of deriving a product in a structured way. An example of the Intrada product family involves the memory dependencies. Different configurations require a different amount of code, heap, and stack memory. Part of the knowledge of how different variants at variation points influence these dependencies has been externalized at Dacolian to documented knowledge (as a series of tables and graphs). The left graph in Figure 3, for example, shows the influence of the selection of different types of matcher variants on the code and data segment. The right graph in Figure 3, on the other hand, shows the influence of a number of reference configurations on the heap and stack memory. These reference configurations consist of the most dominant configuration choices for the dependencies under consideration, according to product type. This knowledge can now be used by a less experienced engineer to determine what choices should be made when memory-size is restricted (e.g. choosing normal matchers, or no matcher at all), or in predicting what the memory usage will be based on matching the product under derivation with the reference configurations.*

As the examples above illustrate, tacit and (to a lesser extent) documented knowledge are an integral part of the knowledge that is needed to derive products. They are responsible for the dependency on expert knowledge and manual labor, which is one of the reasons why the product derivation process is a laborious, time-consuming, and error-prone task that requires a number of iterations before a product is finished [7].

There are two ways to address this problem, i.e. making each step faster (in the sense that they take a shorter amount of time, but produce the same result), or improving the result of each of the steps (which will result in less iterations). Fully, or partially automating assembly, configuration selection, and/or validation is

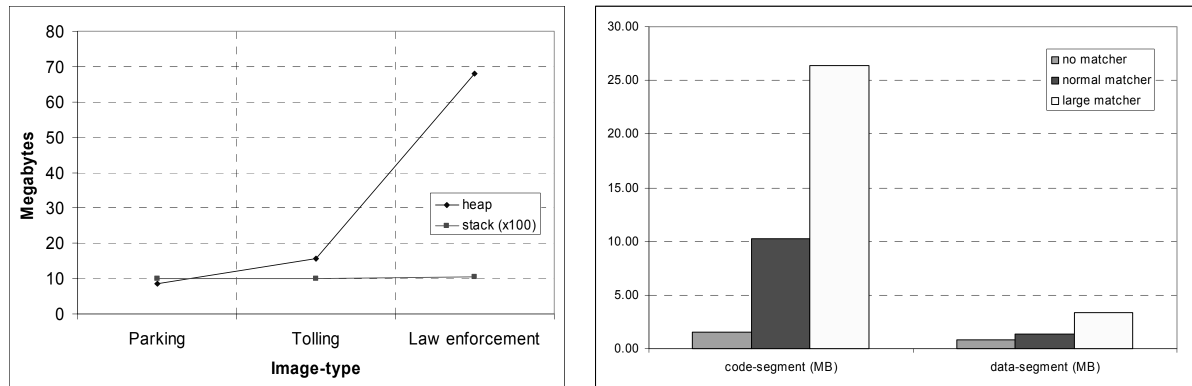

Figure 3. The influence of the most dominant configuration choices on memory usage. This documented knowledge can now be used to predict the memory usage based on a particular selection, and in directing the product derivation process when the restriction on memory usage is not met.

a solution that has the potential to do both, but it requires an underlying methodology that enables it.

It may seem logical that, in order to achieve automation during product derivation, we have to transform all tacit and documented knowledge about dependencies into formalized knowledge. While it is true that transforming tacit and documented knowledge into formalized knowledge reduces the dependency on experts and chance of error during product derivation, this assumption runs into a number of issues. First, this so-called process of externalization [12] suffers from the law of diminishing returns [16], i.e. that the return on investment in externalization effort decreases as the amount of externalized knowledge increases. Second, tacit knowledge is often described as something that is not easy visible or expressible, and therefore difficult to communicate and share with others [12]. Third, depending on the modeling language used, formal specifications can be very hard to maintain, especially if involves an approach where the only way to specify dependencies are uni- or bidirectional in- or exclusions between variants.

2.2. Imprecise or incomplete knowledge

The existence of different knowledge types is not the only aspect that causes problems, however. When we look at Example 3, for example, the documented knowledge only specifies the average behavior for certain types of choices. The graphs do not specify what the exact memory usage in the data segment is when a specific instance of a particular matcher type is selected. Testing the final configuration may therefore reveal that the actual memory usage is higher or lower than initially expected. A second problem when dealing with dependencies is thus the fact that available knowledge can be imprecise or incomplete as well, which prevents a precise formal specification in the first place. This imprecise and incompleteness can be due to the fact that it is not feasible to put enough effort in the externalization. Often, however, the imprecise and incompleteness is due to the complexity of the situation, as we illustrate below:

Example 4: *All Intrada products are built to interpret outdoor video or still images. It is known that variations in image quality require different Intrada configurations in order to deliver high performance. New projects prove that the available knowledge is both imprecise and incomplete. For an image resolution of 2.7 pixels/cm, for example, Dacolian expected that an image recognition rate of 95% correct, with 0.1% error should be feasible. It was discovered, however, that there are complex relations between variation points that are controlled or influenced by aspects such as Signal to Noise ratio, weather conditions, contrast, and country modules used.*

Despite all effort put in by Dacolian, they have not been able to describe all, for configuration relevant, relations between image characteristics and recognition and error rates in a formal or documented way. They currently have to build the system and test it under these conditions to verify the desired value. The best hope so far is a characterization of image quality into categories that are typical for a certain application area. These categories are described by a set of typical images.

Imprecise and incomplete knowledge is especially a problem when the estimated value for a dependency approaches the constraint imposed on the dependency for a product. For example, software engineers can usually be more confident that a constraint of a minimal 80% correct rate (see Figure 2) will be met when the *estimated* correct rate is 90%, than when the estimated correct rate is 81%.

Combined with variation points that influence the value of multiple dependencies, imprecise and incomplete knowledge is a second reason for the necessity of iterations. When, during testing, it proves that certain constraints are not met, the reselection of variants can have an unpredictable effect on multiple dependencies. This can result in multiple trail-and-error iterations. In addition, when it turns out these iterations will not have the apparent positive result, the dependencies have to be weighed against each other (e.g. the correct and error rates in Example 2).

These issues do not mean we can't find a methodology that addresses these issues and enables tool support. In the following section, we discuss how our variability modeling framework COVAMOF deals with these issues.

3. Modeling dependencies in COVAMOF

The idea behind COVAMOF [5][14] is that it provides several views on the variability that is provided by the product family artifacts (see also Figure 4). These views are based on the information in the associated variability model, the COVAMOF Variability View. The main entities in the model are VariationPoints and Dependencies. In this section we focus on how we model dependencies in COVAMOF.

Figure 5 shows the part of the COVAMOF Meta-model that captures the information on the dependencies. The main classes are the Dependency class and the DependencyInteraction class. In the following subsections, we describe both classes in detail. In addition, we describe the classes that build up the information in the Dependency, i.e. the Association and ReferenceDataElement class.

3.1. Dependency

Analogous to the description of dependencies in section 2 and the Introduction, Dependencies represent a system property and specify how the binding of the variation points influences the value of the system property. With the *binding* of a variation point we refer to the selection of variants for that variation point in a specific product. Examples of system properties can be found in the dependencies of the examples 1, 2 and 3,

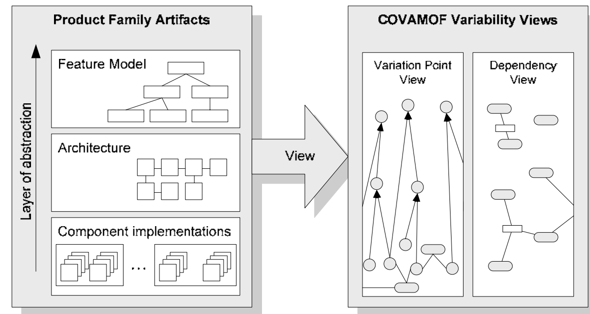

Figure 4. COVAMOF provides views on the variability in the product family artifacts.

where the formal dependency related to pre-compiler directives maps to {true, false}, the error rate maps to [0%, .. ,100%], and the required stack/heap to [0MB, .. , 1024MB], respectively.

As shown in Figure 5, Dependency entities contain six attributes, i.e. the SystemPropertyName, SystemPropertyConstraint, the Associations, the SystemPropertyFunction, the DocumentedKnowledge and the ReferenceData. Below, we explain how the mapping from a binding of the variation points to the dependency values is captured in the variability model by showing the purpose of each of these attributes.

- **SystemPropertyName:** This attribute contains a string representing the name of the system property, e.g. "ErrorRate" or "RequiredStack".
- **SystemPropertyConstraint:** The Dependency entities furthermore specify a valid range of the system property. As this range may vary between product instances, the constraint may contain parameters. An example of a product constraint from Example 2 is "ErrorRate ≤ [MAXErrorRate]". In this case, for each product the MAXErrorRate has to be specified and the actual value of the ErrorRate system property should not exceed MAXErrorRate.
- **Associations:** We refer to the set of variation points that influence the system property of the dependency as the *associated variation points*. For each associated variation point, an Association instance is contained in the Associations attribute. The Association class is described in section 3.2.
- **SystemPropertyFunction:** This attribute contains a (*partial*) function from the binding of the associated variation points to a (estimated) value for the system property. This function is specified formally and is constituted from the formal knowledge that is available on the mapping. For example, the code segment usage of Example 3 can be exactly

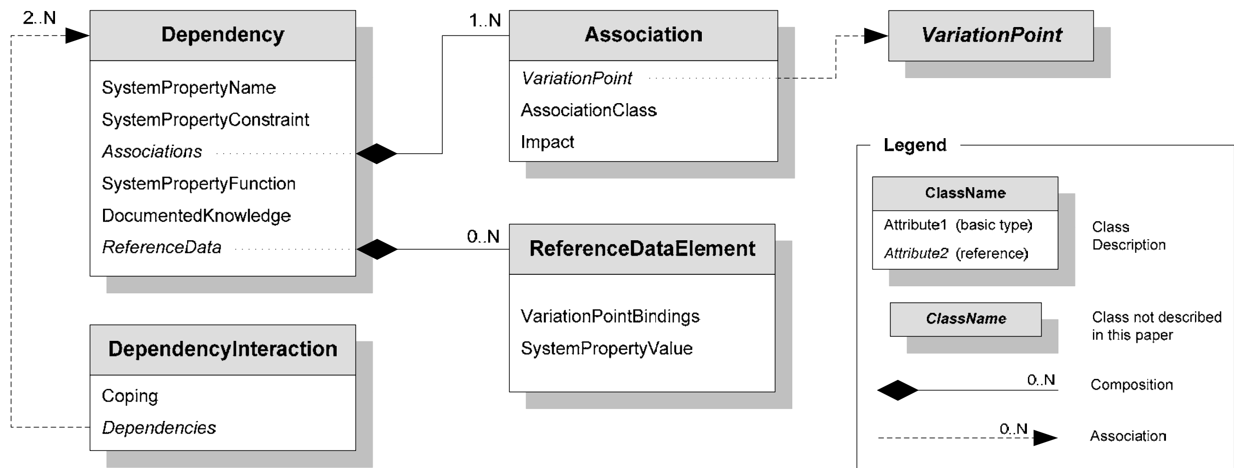

Figure 5. Dependencies in the COVAMOF Meta-model. Dependency entities contain one or more Associations to variation points and zero or more Reference Data elements. The DependencyInteraction entities specify relations between two or more Dependency elements.

calculated from the matcher variants that have been selected (see also Figure 3).
An estimated value can be refined by the information in the Association entities (Section 3.2).

- **DocumentedKnowledge:** This attribute is a set of references to documented knowledge. The contents of a reference can range from a URL, e.g. to a MS Excel file, to the contact information of a product family expert. The engineers can use documented knowledge during the product derivation process to obtain a reasonable first guess of the binding of the variation points. Furthermore, they can use it to estimate the value of a system property based on the binding of the associated variation points. Figure 2, for example, illustrates documented knowledge from the Intrada product family. It contains a graph with data about the system property "ErrorRate" and "CorrectRate", determined by testing the effect of several maximum processing times.
- **ReferenceData:** This attribute contains a set of ReferenceDataElement entities. The purpose and properties of these entities are described in section 3.3.

3.2. Association

For each associated variation point, an Association entity is contained in the Dependency entity. The VariationPoint attribute of an Association refers to a VariationPoint entity in the variability model. A VariationPoint entity refers to the concept of a variation point as we introduced in section 1, and used throughout this paper. The precise description of the VariationPoint entities in COVAMOF is out of the scope of this paper.

COVAMOF distinguishes between three different types of associations, i.e. abstract, directional and logical associations. The type of an association is stored in the AssociationClass attribute of an Association entity. These types relate to the type and completeness of the knowledge about the associations as presented in section 2:

Abstract: On the first level, dependencies contain abstractly associated variation points. The only information product family experts have on abstractly associated variation points is that its configuration influences the dependency values of the dependency. There is no information available on *how* a (re)binding will effect these values.

Directional: On the second level, dependencies contain directionally associated variation points. These associations do not only specify that its configuration influences the values of the dependency, but it also describes some information on *how* the value depends on its binding. The effect of a (re)binding of a directionally associated variation point is not necessarily fully known and documented, and is not specified in a formal manner. The Association entity, however, describes this effect in the Impact attribute as far as it is known to the experts. This information can be used to refine an estimated value from the SystemPropertyFunction of the Dependency.

Logical: On the highest level, dependencies contain logically associated variation points. The effect on the dependency values of the dependency is fully known and is specified as formal knowledge. This

specification is integrated in the SystemPropertyFunction attribute of the Dependency entity.

The system properties of Example 3, i.e. "CodeSegmentSize", "StackSize", and "HeapSize", illustrate all three association types. The total memory required for a product consists of the size of the code segment of Intrada, the maximum stack size and the maximum heap size. The memory required by the code segment is exactly known and calculated upfront from the component selection. All variation points related to the component selection are therefore logically associated in the variability model. For the stack-size, however, it is not precisely known, but software engineers can predict beforehand whether a rebinding of variation points related to the stack size will result in an increase or decrease of required stack space. Therefore, these variation points are directionally associated in the variability model. The effect of variation points on the maximum heap size is largely unknown and can only be determined by testing configurations. These variation points are therefore abstractly associated to the "HeapSize" dependency.

3.3. ReferenceDataElement

In addition to the SystemPropertyFunction and the Associations, dependencies contain reference data about the mapping from the binding of associated variation points to the value of the system property. This reference data is collected during testing, and is defined by a set of ReferenceDataElement entities. Each ReferenceDataElement entity contains a binding of variation points (VariationPointBindings attribute**)** together with the value of the system property for that specific binding (SystemPropertyValue attribute).

Similar to the DocumentedKnowledge attribute of Dependency entities, reference data can be used in two ways. On the one hand, it can be used to find starting points for the derivation process. A reference data element can be selected whose specific SystemPropertyValue is closest to the required value for the product being derived. The bindings of this reference data element can be used as a starting point. On the other hand, the reference data can be used during the derivation process to estimate the value of a system property based on a binding of variation points.

3.4. DependencyInteraction

The variability in industrial product families typically contains a large number of dependencies, each one associated with a number of variation points ranging from one to almost all variation points. Therefore, a lot of variation points are typically associated to more than one dependency. As a result, configuring a variation point in the context of the optimization of one dependency may influence the system property value of all other dependencies that share that variation point. We refer to this mutual influence between dependencies as *dependency interaction*. As the optimal values of those dependencies are often contradicting, such as for the dependencies "CorrectRate" (Example 2), "StackSize" (Example 3), the optimization of dependencies is often a complicated task and involves an extensive amount expert knowledge.

Although the sets of dependencies that interact can easily be generated from the Dependency entities and their Associations, the COVAMOF variability model also explicitly captures DependencyInteraction entities in the variability model. The Coping attribute of DependencyInteraction entities specify, for a set of dependencies, how to cope with the shared associated variation points during product derivation. This textual specification is documented by product family experts and contains a strategy for developing a reasonable first guess during the initial phase, a strategy to optimize the values in the iteration phase, as well as guidance for making trade-offs between interacting dependencies.

4. Benefits

In the previous section, we presented how dependencies between variation points are modeled in COVAMOF. In this section, we discuss the benefits of COVAMOF for different product family engineering activities.

Incremental externalization. COVAMOF explicitly deals with the implications attached to tacit, documented and formalized knowledge (see section 2.1). It does not require a complete and fully formalized model in order to be useful during product derivation. This allows organizations to start with a minimal amount of formalization that can pay off immediately. While in Example 4, Dacolian was unable to formally specify the influences of all the different variation points on the error and recognition rates, specifying *which* variation points influenced the dependencies already provides a good starting point during product derivation.

The benefit provided by a COVAMOF model is that it can be gradually extended when organizational maturity grows and more precise knowledge becomes available, or when more benefits are perceived for the externalization.

Linking knowledge. In addition to incremental externalization, the way in which dependencies are modeled in COVAMOF enables partially formalizing a dependency, and incorporating (links to) documented and tacit knowledge. It provides a central and structured repository for obtaining all product derivation knowledge. This reduces the gaps between tacit, documented and formalized knowledge, and thus the problem of product engineers not being able to find all relevant information.

Reduced expert dependency. In [7], we reported on the high workload and unavailability of experts in software product families. The idea behind knowledge externalization is to reduce the dependency of organizations on experts. Especially when experts have to deal with simple in- and exclude relations that can easily be formalized, time can usually be spent much more wisely elsewhere. In addition, externalization alleviates the vulnerability to knowledge starvation, i.e. loss of important knowledge when experts leave the organization. A further benefit of being able to handle documented knowledge is illustrated in Example 3. In this example, we discussed that tacit knowledge was externalized to documented knowledge in order to enable less experienced engineers to derive products.

Reduced complexity through abstraction. Instead of modeling dependencies between two variants, dependencies in COVAMOF group relations on the level of variation points. They allow specifying dependencies between multiple variants, and provide a more abstract view on relations between choices, thus reducing the overall complexity of the variability model.

Reduced cost, time-to-market and quality through tool support. COVAMOF enables tool support. As we mentioned in section 2, automation can speed up the product derivation process in the sense that different steps can be performed faster by automatically inferring choices. In addition, tools can improve each step as they take over some of the management of the complexity of the variability, for example by improving the overview and combining relevant information. Tools furthermore reduce the amount of human error as they prevent them from forgetting dependencies. This therefore also allows the product derivation process to achieve the same result with less iterations.

Less iterations. Tool support is not the only way in which the number of iterations can be reduced, however. By explicitly dealing with the complex dependencies, product experts also earlier spot when there is a safe margin for certain dependencies, or when certain configurations approach the constraints imposed upon them (see e.g. Example 2 and 3). Instead of requiring engineers to build the product first, otherwise unexpected problems can be addressed upfront, thus saving iterations. Indirectly, this also leads to a more predictable derivation process.

Recording and using product derivation data. The reference data and dependencies enable storing useful product derivation knowledge. The test results for a dependency in a particular configuration can for example, be reused to know or estimate the value of a dependency in a new configuration. Multiple data points can furthermore be generalized to variants with specific properties, and the results can be stored in the dependencies for use during product derivation, or in component development. The documented knowledge in Figure 2 is an example of reference data that is generalized to documented knowledge. Data can furthermore be used to evaluate the provided variability with respect to actual use.

Static and dynamic product derivation strategies. A product derivation strategy specifies the order in which choices and constraints are set during product derivation. Tool support enables formulating both static and dynamic product derivation strategies. Based on profiling of historical data collected during product derivation, such as which dependencies proved problematic, the derivation process can be improved with a static strategy, e.g. by suggesting a different order of choices. The process can furthermore be improved with dynamic strategies that change the suggested order on-the-fly by taking, for example, the impact of choices and number of dependencies between variation points into account.

5. Conclusion

Variability modeling plays a key role in product derivation issues we identified in earlier work [7]. Many of these problems are related to dependencies, i.e. relations between a set of variation points. The dependencies specify a system property whose value is based on the selection of variants at the different variation points. In [15], we concluded that most existing variability modeling approaches (e.g. [1][2][3][4][10][13][17]) only address the formalization of dependencies. In this paper we showed that the problems that exist in practice are not fully addressed. In practice, dependencies are more complex than simply requiring or excluding another variant, and knowledge related to these complex dependencies is often imprecise or incomplete. In Section 2, we illustrated these practical problems with a number of examples. We furthermore related these problems to different types and forms of knowledge

that exists, i.e. tacit, documented and formalized knowledge, and imprecise and incomplete knowledge.

In COVAMOF, we address these issues by modeling dependencies as first-class citizens. With first-class dependencies, COVAMOF provides a number of benefits, such as being able to incrementally specify the variability model, linking different types of knowledge, reducing expert dependency, reducing complexity, reducing cost, quality and time-to-market issues through tool support, reducing the number of iterations, being able to record and use product derivation data, and being able to create static and dynamic product derivation strategies that optimize the product derivation process (see Section 4).

COVAMOF is supported by a tool-suite. This tool suite consists of a number of plug-ins for Microsoft Visual Studio .NET [11]. The suite integrates the COVAMOF models with the product family artifacts (e.g. source code and binaries), and enables both textual and visual modeling of the provided variability. It furthermore provides facilities for product derivation and variability assessment [9]. For more details on this tool suite, see [5].

6. References

[1] T. Asikainen, T. Soininen, T. Männistö, “A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families”, *5th Workshop on Product Family Engineering (PFE-5)*, Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 225-249, May 2004.

[2] F. Bachman, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, A. Vilbig, “Managing Variability in Product Family Development”, *5th Workshop on Product Family Engineering (PFE-5)*, Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 66-80, May 2004.

[3] M. Becker, “Mapping Variability’s onto Product Family Assets”, *Proceedings of the International Colloquium of the Sonderforschungsbereich 501*, University of Kaiserslautern, Germany, March 2003.

[4] M. Clauss, “Modeling variability with UML”, *GCSE 2001 - Young Researchers Workshop*, September 2001.

[5] COVAMOF website, http://www.msinnema.nl/covamof.

[6] Dacolian B.V. website, http://www.dacolian.com.

[7] S. Deelstra, M. Sinnema, J. Bosch, “Product Derivation in Software Product Families; A Case Study”, *Journal of Systems and Software*, Vol 74/2, pp. 173-194, January 2004.

[8] S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, “Experiences in Software Product Families: Problems and Issues during Product Derivation”, *Proceedings of the Third Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes on Computer Science Vol. 3154 (LNCS 3154)*, pp. 165-182, August 2004.

[9] S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, COSVAM: A Technique for Assessing Software Variability in Software Product Families, *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pp. 458-462, September 2004.

[10] H. Gomaa, M.E. Shin, “Multiple-View Meta-Modeling of Software Product Lines”, *8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society 2002*, ISBN 0-7695-1757-9, pp. 238-246, 2002.

[11] Microsoft Visual Studio .NET website, http://msdn.microsoft.com/vstudio.

[12] I. Nonaka, H. Takeuchi, *The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation*, Oxford University Press, New York, 1995.

[13] R. van Ommering, “Building Product Populations with Software Components”, *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, May 2002.

[14] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, “COVAMOF: A Framework for Modeling Variability in Software Product Families”, *Proceedings of the Third Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes on Computer Science Vol. 3154 (LNCS 3154)*, pp. 197-213, August 2004.

[15] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, “Managing Variability in Software Product Families”, *Proceedings of the 2nd Groningen Workshop on Software Variability Management*, December 2004.

[16] W.J. Spillman, E. Lang, *The Law of Diminishing Returns*, 1924.

[17] S. Thiel, A. Hein, “Systematic integration of Variability into Product Line Architecture Design”, *Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2)*, August 2002.