

The COVAMOF Derivation Process

Marco Sinnema, Sybren Deelstra, Piter Hoekstra

University of Groningen, 9700 AV Groningen
Email: {m.sinnema|s.k.deelstra}@rug.nl,
p.s.hoekstra@student.rug.nl

Abstract. The design, usage and maintenance of variability, i.e. variability management, is a very complex and time-consuming task in industrial product families. The COVAMOF Variability Modeling Framework is our approach to enable efficient variability management. As a practical realization of COVAMOF, we developed the COVAMOF-VS tool suite, which provides several variability views on C#, C++, Java, and many other types of projects in Microsoft Visual Studio .NET. In this paper, we show how COVAMOF facilitates an engineer during product derivation, and what benefits are gained by it.

1 Introduction

A software product family is an approach to reuse that involves creating a collection of similar software systems from a reusable set of software artifacts [1][2][13]. As product families promise order of magnitude improvements in quality and productivity of software development, more and more organizations are adopting this approach [7].

The ability to derive distinct products from a product family is supported 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. On the one hand, variability is enabled through variation points, i.e. the locations in the software that enable choice at different abstraction layers (features, architecture and implementation layer). Each variation point is associated with a number of options (represented by variants or values). On the other hand, the possible configurations that can be build with these variation points are restricted due to dependencies that exist between variants, and the restrictions that are imposed upon these dependencies.

Product derivation in software product families involves making a large number choices (up to ten thousands [4]) at variation points in the available artifacts (at feature [6], architecture, and implementation level). These large numbers make it hard to manage all variation points by humans. Even though it is absolutely necessary to handles these numbers effectively, however, there are even more complicated issues in managing variability.

In Sinnema et al. [11], for example, we discussed variability management issues that are the result of *complex dependencies*. Consider, for example, a design where the maximum processing time of data is constrained. The configuration of many variation points in the design influence this maximum processing time. The complex

dependency (or value of the property) maximum processing time is the result of the combination of variants that is selected at each of these variation points. Determining the value of this property is typically much more complicated than handling simple logical in- and exclusions.

The complications for complex dependencies are primarily caused by the fact that the knowledge that is available for these dependencies is often tacit (in minds of experts) [8], incomplete, and imprecise. In addition, complex dependencies suffer from dependency interaction, i.e. that due to the fact that several variation points are involved in multiple dependencies, trying to meet the constraints on a particular dependency value may influence other dependency values as well. For an extensive discussion on these issues, see Sinnema et al. [11].

Existing software variability modeling approaches and tools are inadequate to handle these variability management issues [10]. In response, we developed our own variability modeling framework COVAMOF [10]. This framework consists of modeling facilities that model the variation points and dependencies uniformly over different abstraction levels (e.g. features, architecture and implementation), and as first-class citizens. As part of our framework, we developed the COVAMOF-VS tool suite. The COVAMOF-VS tool suite is a set of Add-ins for Microsoft Visual Studio .NET [9]. It is designed for creating variability models of a product family, and using these models for configuration of individual products.

In earlier work [10], we focused on presenting the modeling concepts in COVAMOF. Before a modeling framework can be used to manage variability in a product family, however, it should be clear how the model needs to be created, maintained, and used. The purpose and contribution of this paper lies in showing how such a model is used during product derivation, and how COVAMOF facilitates a software engineer when he/she needs to derive a product from the product family.

In order to provide a concrete description of the COVAMOF Derivation Process, we discuss this process according to how the tool-suite is used, and provide initial results of the validation of COVAMOF. To present the derivation process in a concise manner, we first present a brief example that is used for illustrating our message. In section 3, we provide an introduction to COVAMOF and the COVAMOF-VS tool-suite. In section 4, we show the practical benefits of COVAMOF according to different steps in the derivation process. We revisit the core issues we mentioned above in section 5, and show how they are addressed by our approach. In section 6, we present a summary of empirical validation we gained by applying our approach at an industrial organization. We conclude this paper in section 7.

2 Example case

To illustrate COVAMOF, we use an example in this paper (see also Fig. 1). This example originates from a product family that is built by the same organization from which we use empirical results to validate COVAMOF. The example involves a product family that is built on top of the product family we use in the validation. Most of the variability in this example has been abstracted away, or simply left out, so that the

reader can easily grasp the example. The purpose of this small example is not to validate COVAMOF, but to explain some of the basic concepts.

The example involves a product family in the domain of license plate recognition on handhelds (PDAs, and smartphones). License plate recognition on handhelds involves automatically recognizing license plates on images captured through a camera. These license plates are on- or offline checked against a white/blacklist using communication channels such as WAN, Bluetooth, GPRS, or cable. The products in this example are modules that a customer uses to rapidly build their own handheld recognition application. The modules for example offer a view-port for displaying camera images, an interface for a license plate recognition engine, as well as communication services.

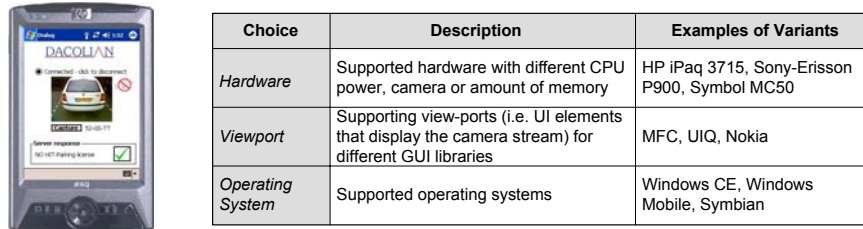


Fig. 1. Example product family. This figure shows a picture of the HP iPaq 3715 running the application (left) and some examples of the variability (in terms of choices) provided by the product family (right).

3 COVAMOF

In order to support product family engineers, we developed the COVAMOF Variability Modeling Framework and the associated tool-suite for Microsoft Visual Studio .NET [9] (COVAMOF-VS). Although the focus of this paper is to explain how COVAMOF helps software developers to choose the right variants and settings that satisfy the constraints and functionality required for a new product, we first need to explain the basic concepts of COVAMOF. In the following sections, we discuss these concepts according to the two main views provided by the COVAMOF-VS tool-suite.

3.1 COVAMOF, Introduction

As we mentioned in section 1, COVAMOF is a framework for modeling variation points and dependencies uniformly over different abstraction levels (e.g. features, architecture and implementation), and as first-class citizens. This framework enables providing different views on the variability provided by product family artifacts. At this point, the tool-suite COVAMOF-VS provides two main graphical views, i.e. the variation point view, and the dependency view (see Fig. 2).

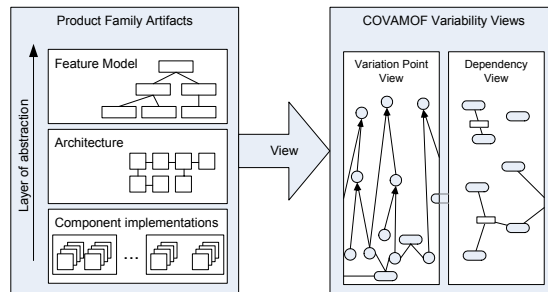


Fig. 2. COVAMOF Views. By treating dependencies and variation points as first-class citizens, COVAMOF enables providing a separate dependency view on the variability provided by a product family

To provide these views, COVAMOF-VS maintains an integrated variability model. This model is constructed by reading variability information from the collection of files in the active Solution in Microsoft Visual Studio. The active Solution in MS Visual Studio contains the artifacts that constitute the software product family. The Solution can contain very different artifacts, e.g. XML-based feature models, start-up parameter specifications and C# source files.

The variability information in these artifacts is either directly interpreted from language constructs (such as `#ifdef`), or is based on special constructs from the COVAMOF variability language XVL. While we eventually strive for extending a programming language with these constructs, variability information in code files is currently inserted as comments. These constructs are inserted in the solution artifacts at the time they are under development, or afterwards using a variability modeling process as discussed in section 4.

The extraction of variability information is done by plug-in components that register themselves on one or more file types. These components convert the extracted variability to parts of the COVAMOF variability model. They are also responsible for feeding additions and changes in the views directly back into the files of the MS Visual Studio Solution.

Once the model is constructed, it can be viewed graphically in the variation point and dependency view. The variation point and dependency view each serve a specific purpose. In the following two sections, we discuss the entities in the COVAMOF model (see Fig. 3) according to these different views. In section 4, we discuss how these views are used during product derivation.

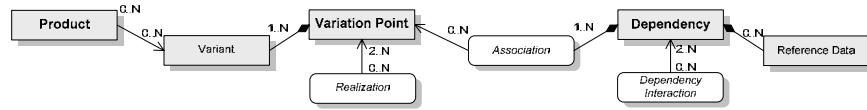


Fig. 3. The COVAMOF Meta-model

3.2 COVAMOF, The Variation Point View

The main purpose of the variation point view is to show to the engineer, which choices are available at different abstraction layers, how they realize each other across layers, and how choices depend upon each other. The variation point view contains the following entities: *Variation Point*, *Variant*, *Realization* and *Dependency* (see also Fig. 3). These entities are illustrated with a screenshot from the example case (see Fig. 4).

Variation Point: *Variation Points* in COVAMOF represent the location at which a choice is provided by the product family. A *Variation Point* entity has a number of properties, for example for storing the abstraction layer the choice is located in (abstraction layer), the moment in the lifecycle at which the choice is bound (binding time), and storing the reason why a choice is provided (rationale). The different options that are available for a choice are represented by a value, or by a set of variants that are associated to the *Variation Point*.

Variant: *Variant* entities represent the options that are available at a *Variation Point*. While values are used to represent parameters, *Variants* can represent anything from an object or class, to a file, or a code-block. The effectuation property of a *Variant* specifies the effectuation actions that should be executed in the product family artifacts when the variant is selected. Examples of such effectuation actions are the generation of a configuration file, the setting of compiler directives and the specification of libraries that have to be linked.

Example. The screenshot in Figure 4 shows a *Variation Point* 'Supported OS', with *Variants*, 'Symbian', 'Windows CE', and 'Windows Mobile 2003'. The *Variation Point* is situated in the feature layer. The information displayed here tells that a software engineer has the option to choose to derive a product that runs on one of these three operating systems.

Realization: The *Variation Point* entities are hierarchically organized over abstraction levels. In this hierarchy, variation points in lower levels of abstraction realize the variability on a higher level abstraction. *Realization* relations specify *rules* that determine which variants or values at variation points at lower levels should be selected in order to realize the choice at variation points at higher levels.

The purpose of explicitly modeling these relations in the variability model is two-fold: first, as the realization relations capture the knowledge about *how* variation

points realize other variation points, COVAMOF-VS can automatically infer choices on a higher level of abstraction to choices on a lower level of abstraction, reducing the human effort to configure products. Second, the hierarchical organization of variation points structures the variability in such a way that a software engineer does not have to consider all the variation points at once. Instead, he can just focus on one relevant subset of all the variation points. For example, he can just focus on the high level variation points that realize the overall product family variability, or just focus on the lower level variation points that together realize only one aspect of the product line variability. Therefore, it allows humans to manage the complexity caused by large numbers of variation points in industrial product families.

Example. In Figure 4, the *Realization* relation from 'CameraInterface Implementation' to 'Supported OS' specifies that different implementations for the Camera Interface realize the ability to choose between supported operating system. The rules at this Realization (not shown in Figure 4), specify which CameraInterface Implementation should be selected when a particular OS is chosen. An observing reader may note from other Realization relations that there is a strong coupling between Operating System, Device, and CameraInterface implementation.

Dependency: The COVAMOF variability model captures both simple and complex dependencies (see also section 1). These dependencies are represented as *Dependency* entities in the variability model. They specify a mapping from the configuration of a set of *Variation Points* to a *value* in a specific 1-dimensional domain. In plain English, this means that if you select *Variants* or a value at *Variation Points* (called the configuration), the *Dependency* maps this selection to a *value* for a property (the target domain) such as maximum processing time or memory consumption. We refer to the set of *Variation Points* whose configuration influences the value as the *associated variation points*. The model combines two ways of capturing the knowledge about *how* the configuration of the associated variation points maps to a value in the target domain, i.e. by *Association* entities and *Reference Data* entities.

Association: The first way to capture this knowledge is by *Association* entities, which are part of a *Dependency*. These *Associations* refer to the *Variation Points* whose configuration affects the *value* of the *Dependency*. Each *Association* corresponds to the relation with one *Variation Point*. COVAMOF distinguishes between three types of *Associations*. In increasing order of completeness of the knowledge about the associations, these are Abstract, Directional, and Logical *Associations*. An Abstract *Association* means that experts only know that a relation between the *Variation Point* and *Dependency* exists, but have no way of predicting the effect of selecting a different *Variant* or value at the *Variation Point*. A Directional *Association* means that experts have an idea about the direction in which the value of a dependency will change when a different value or *Variant* is selected. In case of Logical *Associations* experts know exactly how the value of a dependency will change when a different *Variant* or value is selected. Product family experts can enrich the *Associations* with textual hints that explain how variation points can be reconfigured in order to obtain a certain *value*.

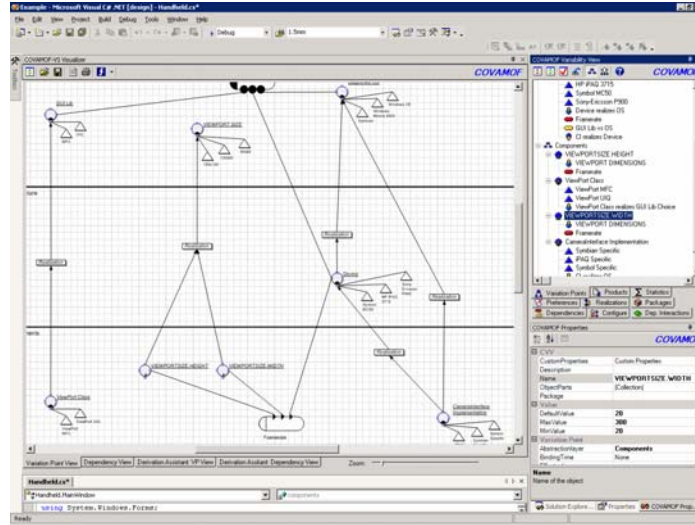


Fig. 4. A screenshot of the example case in COVAMOF-VS. This figure shows some of the variation points and dependencies that are involved to provide the variability of the example case we discussed in section 3

The type of associations of a Dependency influences the type of the Dependency. In COVAMOF, we distinguish between Statically Analyzable and Dynamically Analyzable Dependencies. For Statically Analyzable Dependencies, the exact value can be calculated before running the system, which requires Logical Associations. The exact value of Dynamically Analyzable Dependencies can only be determined by running and measuring the system, which is the case if the Dependency contains Abstract and Directional Associations.

Reference Data: The second way of capturing the knowledge about how the configuration of the associated variation points maps onto the value of a dependency is modeling reference data. Reference Data entities contain measurements of the value of a Dependency for specific configurations of the associated Variation Points. These measurements originate from tests of products that have been derived from the product family. These Reference Data entities can in turn be analyzed to determine how single Variation Points affect the Dependency. The results of this analysis are in turn stored in the Associations.

Example. In Figure 4, the Dependency 'Framerate' is associated to the three Variation Points 'Device', 'VIEWPORTSIZE.HEIGHT', and 'VIEWPORTSIZE.WIDTH'. The first Variation Point refers to the handheld device that can be chosen, while the latter two Variation Points refer to configuration parameters that set a view-port size. The information displayed in Figure 4 therefore shows that the 'Framerate' of the application, i.e. the number of times/second an image is displayed, is dependent on the size of the view-port, and the selected handheld device. The Directional Associations specify how the frame-rate depends on these Variation Points (not shown in Fig. 4).

3.3 COVAMOF, The Dependency View

The dependency view contains *Dependencies* and *Dependency Interactions* (see Fig. 3). The main purpose of this view is to show how dependencies interact with each other, and how an engineer can cope with these interactions.

Dependencies: *Dependencies* in the dependency view are representations of the same dependencies that are represented in the variation point view. In the dependency view, however, they only show which dependencies exist, and do not show the relationships between variation points.

Dependency Interaction: In the introduction, we explained that dependency interaction occurs when variation points are part of multiple dependencies. Although the sets of dependencies that interact can be generated from the dependencies and their associations, COVAMOF also explicitly captures *Dependency Interaction* entities in the variability model. These entities allow a software engineer to 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. It contains a strategy for developing a reasonable first guess during the initial phase, and a strategy to optimize the values in the iteration phase of product derivation.

4 COVAMOF Derivation Process

Product derivation is the construction of a software product that is built by selecting and configuring product family artifacts. With COVAMOF, products are derived by the COVAMOF Derivation Process. This process allows organizations to gain maximum benefit from COVAMOF and its associated tool support. This COVAMOF Derivation Process is divided into four steps, i.e. Product Definition, Product Configuration, Product Realization, and Product Testing. As visualized by Fig. 5, the last three steps can occur in one or more iterations. The following subsections describe these four steps respectively.

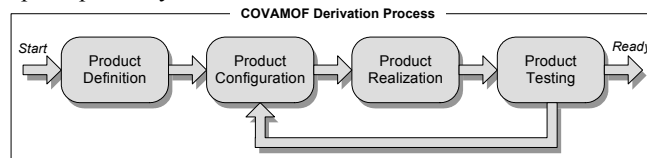


Fig. 5. The COVAMOF Derivation Process. This iterative process breaks down into four steps, i.e. Product Definition, Product Configuration, Product Realization and Product Testing

4.1 Product Definition

The first step of the COVAMOF Derivation Process is called Product Definition. In this step, the engineer creates a new *Product* entity in the variability model. In response, COVAMOF-VS stores this product in the active Solution of Visual Studio. The properties of Product entities are the customer, a unique name for the product, and variation points that have been bound for the product. The latter are described by bindings of variation points, i.e. combinations of variation points together with their selected variants or parameter value. The engineer can directly fill in the customer and name property. For the bindings of the variation points, one or more iterations of the Product Configuration step are required.

4.2 Product Configuration

In order to start the Product Configuration step, the engineer selects the *Product* entity from the available products dropdown menu in the COVAMOF-VS toolbar. From that point, COVAMOF-VS is in configure mode and additional configuration information about the product at hand is shown in both variability views.

When the variation point view of COVAMOF-VS is in configure mode, the variation points that are bound for the product at hand are marked by a different color. During the Product Configuration step, the engineer binds one or more variation points to new values or variants based on the customer requirements. In order to bind these variation points, he marks the new variants or specifies the new values in the variation point view. The order in which variation points are bound is dynamically determined by the realization relations and dependencies in the variability model.

Meanwhile, the binding of a variation point is effectuated by creating or updating the relation between the *Product* entity and the *Variation Point* entity. Each variation point that is bound triggers the inference engine and the validation engine to work:

- **Inference Engine:** As described in section 3.2, the rules of *Realization Relations* define how variation points on a lower level of abstraction realize variation points on a higher level of abstraction (see also Fig. 4). These rules are used by the inference engine to automatically bind variation points on a lower level of abstraction. The binding of these variation points is based on the binding of the variation points on a higher level of abstraction. For each rule that the variation point is involved in, the consequences are recursively determined.
- **Validation Engine:** After a variation point is bound by the engineer and the inference engine has worked, the validation engine automatically checks whether no dependencies have been violated. The rule of each statically analyzable dependency is checked and for each dynamically analyzable dependency the reference data elements are checked. Based on the rules, the associations and the reference data, the dependencies in the variation point view are provided with the new estimated *value*. Any violations of these values with respect to the required values are immediately fed back to the engineer by marking the dependencies in the variation point view with the color red.

The customer requirements can be separated into functional requirements and non-functional requirements. Therefore, the engineer basically has two main concerns during product configuration. First, the right features and components should be selected so that the functional requirements are met. Second, the configuration is tuned and adapted to meet the non-functional requirements. The *Realization Relation* and *Dependency* entities in the variability model support the engineer in configuring a product that meets the functional as well as the non-functional requirements. *How* these two entities can be used is explained below. Note that in practice, the engineer has to take both into consideration at the same time.

- **Realization relations:** In order to select the right features and components, the engineer binds variation points in the feature layer. The variation points in the architecture layer can be bound using rules of the *Realization Relations* between the feature layer and the architecture layer. In case the inference engine was unable to automatically bind these variation points in lower layer of abstraction, the engineer has to bind them manually. Similarly, the variation points in the component layer can be bound by using the realization relations between the component layer and the architecture and feature layer.
- **Dependencies:** The dependency entities in the variability model support the engineer in binding variation points in such a way that the product is consistent and meets the non-functional requirements. When Product Configuration starts, the engineer specifies, for each of the dependencies the specific value or range that is required for the product at hand. The variation point view of COVAMOFVS in configure mode shows the (estimated) value of each of the dependencies together with the required value or range. When the actual value is outside the required range, i.e. the *Dependency* is violated, the dependency in the variability view is colored red.

The goal of engineers is to (re)bind the variation points in such a way that none of the dependencies are violated. During iterations, the engineer therefore has to shift the *value* of each violated *Dependency* into the required range. The information in the *Associations* and the *Reference Data* of the *Dependency* entity are used to determine *how* the current *value* can be changed in order to meet the required range.

As Logical *Associations* specify exactly how the (re)binding of a variation point changes the *value*, this formalized knowledge can easily be used to change the value of the *Dependency*, and the effect is immediately visible in the variation point view. The documented knowledge in the Directional *Associations* can be used to increase or decrease the *value* in the right direction. However, a Product Test (section 4.4) is required to determine the new value of the *Dependency*. How Abstract *Associations* can be used can only be determined by any reference data available. Otherwise, the engineer has to use the tacit knowledge of an expert or even trial-and-error to change the *value* of the *Dependency*.

Note that the process of changing the *values* of *Dependencies* is a very complicated task. This is particularly due to the interaction between *Dependencies* like we described in the introduction. In such situations, the engineer uses the information of the corresponding *Dependency Interaction* entities in the dependency view (section 3.3). This information helps the engineer in making reason-

able trade-offs between system properties and a strategy to accomplish the acceptable *values*, for example, which *Dependency* should be resolved first and which should be resolved later in the process.

Usually, the focus during the initial iteration of the COVAMOF Derivation Process is on satisfying the functional requirements, and, as the product functionality becomes more and more fixed, the focus gradually shifts to satisfying the non-functional requirements. This does not imply that functional requirements are more important. Functional properties are usually the easy aspect during product derivation, and generally have to be known in order to say something meaningful about the non-functional properties.

When the engineer is unable to bind additional variation points without testing the configuration at hand, the COVAMOF Derivation Process goes to the next step, Product Realization.

4.3 Product Realization

In order to get a complete software product, the *Product* has to be realized in the product family artifacts. In COVAMOF-VS, the product is realized by pressing the Realize button in the toolbar of COVAMOF-VS. As a result, COVAMOF-VS executes the effectuation actions for each of the variants that are selected for the Product entity (see also section 3.2).

Thereafter, Visual Studio builds the active Solution, resulting in the binaries that are used together with the configuration files to test the product in the next step.

Example. The example in Figure 4 contains the *Variation Points* 'VIEWPORTSIZE.HEIGHT, and 'VIEWPORTSIZE.WIDTH'. When during the Product Configuration step these variation points both have been bound to 120, the following start-up parameters are specified in the configuration file `mobileapp.ini`:

```
[Viewport]
Height=120
Width=120
```

4.4 Product Testing

The goal of the testing phase is to determine whether the product meets both the functional and the non-functional requirements. When, during testing, the realized product appears to satisfy all requirements, the software product can be packed and shipped to the customer. Otherwise, one or more additional iterations of a Product Configuration and Product Realization steps are required.

In any case, the *values* of the dynamically analyzable *Dependencies* that have been determined during the test are fed back into the variability model as *Reference Data*. In this way, the COVAMOF variability model is gradually enriched and improved to provide better estimated *values* during Product Configuration steps in the future.

5 Benefits

COVAMOF and COVAMOF-VS were created to address the issues that are experienced by software engineers in many industrial families (see e.g. Deelstra et al. 2005 [4]). In section 3 and 4, we discussed the main entities in the COVAMOF meta-model (see also Fig. 3), and showed how they provide practicable benefits during product derivation. In this section, we summarize how the combination of COVAMOF and COVAMOF-VS addresses the variability management issues we discussed in the introduction.

1. Effectively handling complex dependencies. Instead of modeling dependencies between two variants, dependencies in COVAMOF group relations on the level of variation points. This allows specifying complex dependencies between multiple variants that would otherwise translate into a large amount of dependencies between variants. This grouping furthermore provides a more abstract view on relations between choices, thus reducing the overall complexity of the variability model. As the dependencies are first-class, COVAMOF is also able to provide a separate dependency view that shows the interaction between them.

2. Ability to use imprecise, tacit and documented knowledge. In addition to the formal specification of the variability model, COVAMOF explicitly deals with tacit, documented, and formalized knowledge through the different types of dependencies. Although tacit knowledge is not represented in a COVAMOF model (otherwise, it wouldn't be defined as tacit), COVAMOF deals with tacit knowledge by enabling references to the experts that possess this knowledge (e.g. through names, phone numbers, etc.). This may seem silly, but the importance should not be underestimated: it allows (less experienced) engineers to call-in the right assistance.

Documented and formalized knowledge are handled in several ways. The *Reference Data* and *Associations* at *Dependencies* enable storing useful product derivation knowledge, e.g. in terms of links to documents, graphs, and formulas. The test results for a *Dependency* in a particular configuration can be reused to know or estimate the value of a *Dependency* in a new configuration. Multiple data points can be generalized to variants with specific properties, and the results can be stored into *Associations*. As we explained in section 4, these *Associations* are used during the COVAMOF Derivation Process to estimate the impact of choices on dependencies.

All these facilities allow organizations to start with a minimal amount of formalization that can pay off immediately. This model can be gradually extended when organizational maturity grows and more precise knowledge becomes available, or when more benefits are perceived for the externalization.

3. Dependency interaction. To reduce the expert involvement and number of iterations, the problem of dependency interaction is addressed by explicitly capturing *Dependency Interaction* entities. These entities specify a strategy that suggests to an engineer, which steps he/she should follow in trying to satisfy a particular set of interacting dependencies.

6 Validation

We are involved in a case study that introduces COVAMOF in an industrial setting. The subject of this case study is the Intrada product family of Dacolian B.V. Intrada is an industrial product family for complex intelligent traffic systems such as license plate reading, tolling, and parking applications [3]. At this point, the variability model has been integrated into the software product family. Dacolian B.V. is in the process of collecting data with respect to the use of COVAMOF, such as number of man-hours and iterations saved, as well as data on the experience of engineers with the tools. For each product derived, they keep track of which variation points are bound in the iterations of the COVAMOF Derivation Process, and how many hours are required to perform each step.

Although we cannot yet present the definitive quantitative results of this case study, Dacolian B.V. already provided interesting qualitative observations. First of all, Dacolian B.V. is convinced that COVAMOF saves an enormous amount of man-hours. Before the introduction of COVAMOF, the derivation of a typical product from the product family required about one day of work. Now, their engineers can derive the same product within a quarter of an hour. Below, we provide some detailed qualitative observation we received from Dacolian B.V.:

- **Number of iterations reduced:** “The engineers are provided with much more useful information about the choices they have to make and dependencies they have to consider. As a result, the decision making of the engineers has improved and they are able to make better estimations for the variants and values that have to be selected. Therefore, the iterations required for a typical product derivation process has been reduced from 10-12 iterations to 0-2 iterations after the introduction of COVAMOF. This reduction of iteration, together with the automatic inference of bindings, resulted a dramatic reduction of required man-hours”.
- **Finding Conflicts:** “Before we were able to implement the variability model of the product family, we had to externalize the variability information from the product family artifacts and the experts. During this externalization process, engineers and experts from our organization found many (previously unknown) conflicts in their own artifacts. This has, and will save us a lot of unforeseen problems”.
- **Development and Evolution:** “A substantial part of the products of Dacolian B.V. are based on the automatic recognition of license plates. One key variation point for this family is the collection of countries of which license plates can be recognized. Before the introduction of COVAMOF only product line experts were able to extend the number of supported countries, i.e. implementing a new variant for this variation point. After COVAMOF has been implemented, with all major dependencies externalized, also engineers that are not involved in the product family are able to extend the number of supported countries. This reduced the workload of the product line experts”.

7 Conclusion

Software variability management is an important factor in the success of a product family. It is also a complex task, where key issues such as handling complex dependencies, dealing with imprecise, tacit, and documented knowledge, and dependency interaction, are inadequately addressed by existing approaches [10].

In this paper, we discussed the COVAMOF Product Derivation Process. We described this process using the technical realization of COVAMOF in the form of the COVAMOF-VS tool-suite. We have shown how the different elements of COVAMOF address the key variability management issues during product derivation, and supported these claims with empirical results from an industrial application of COVAMOF.

8 References

1. 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
2. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7, 2001
3. Dacolian B.V.: <http://www.dacolian.nl>
4. Deelstra, S., Sinnema, M., Bosch, J.: Product Derivation in Software Product Families; A Case Study, Journal of Systems and Software, Volume 74(2), pp. 173-194, January 2005
5. Deelstra, S., Sinnema, M., Nijhuis, J., Bosch, J.: 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
6. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021, 1990
7. v.d. Linden, F.: Software Product Families in Europe: The Esaps & Café Projects, IEEE Software, Vol. 19, No. 4, pp. 41-49, 2002
8. Nonaka, I., Takeuchi, H.: The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation. Oxford University Press, New York, 1995
9. Microsoft Visual Studio .NET: <http://msdn.microsoft.com/vstudio>
10. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: 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
11. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Modeling Dependencies in Product Families with COVAMOF, Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, March 2006
12. Spillman, W.J., Lang, E.: The Law of Diminishing Returns, 1924
13. Weiss, D. M., Lai, C.T.R.: Software Product-Line Engineering: A Family Based Software Development Process, Addison - Wesley, ISBN 0-201-694387, 1999