In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR),
Tampere, Finland, 24-26 March, 2004, pp. 261-269.

Tool Support for Traceable Product Evolution

P. Lago[*], E. Niemelä[♣], H. Van Vliet[*]
[*]Vrije Universiteit, Amsterdam, The Netherlands, [patricia | hans]@cs.vu.nl
[♣]VTT Technical Research Centre of Finland, Oulu, eila.niemela@vtt.fi

Abstract

In software product families, the full benefit of reuse can only be achieved if traceability of requirements to architecture, components and further down to source code is supported. This requires automated tool support for tracing from the abstract features of the product family to a set of concrete features and source code of family members.

We extended a commercial software tool to support top-down as well as bottom-up traceability in product families, from the family feature map all the way down to implementation files. At the code level, both newly developed and commercial-off-the-shelf components are accommodated. The tool has been validated by (bottom-up) filling the tool's reuse base with features, components, documentation files, etc. from six related products in the Next Generation Network service domain, and next deriving a seventh product from this reuse base.

1 Introduction

The life cycle of software intensive systems is continuously shortening due to the acceleration of technology development and cutting time-to-market. Modern software systems, most of them distributed and embedded in our every-day operational environment, are complex systems that require systematic methods to develop and maintain them. They also require tools that allow tracing design decisions, made in early phases of development, to their realization in final products. Especially if products are members of a product family developed in several places and composed from existing proprietary and third-party components. The larger the software systems, the more crucial their variability management is. This is due to the increasing size and complexity of these systems and the wish to reduce software development costs through lengthening the use of existing artifacts, i.e. features, architecture and components.

Diversity in product features, resulting from the needs of market segments and technologies used, has brought out several problems in software development and evolution: inability to describe product variants clearly, to understand descriptions of product variants, and to maintain product variations. Although several methods are introduced for modeling variability as feature graphs [11] [16] [10] [17] and variation points [18] [6] [3], there is still a lack of techniques and tools for variability management [12]. Frequent changes in product features in particular require new methods and techniques to handle diversity at different abstraction levels and during the evolution of a product family and family members. This means management of abstract features at the product family level, variation points in product family architecture, configuration rules for assembling products and built-in reconfiguration mechanisms in systems for changes required in delivered products.

The development of a product family is time-consuming and expensive, and repayment can happen only if its features, architecture and components are used several times, in the best case during the life cycle of a family member while developing, installing and upgrading a software system. In practice, however, it seems to be unsolved 1) how to trace defined product features through architecture and components to source code, and 2) how to become convinced that a proper set of variable features is incorporated in a product.

In order to solve the above mentioned two problems there are several technical issues to be considered. First, we need a method and supporting tool to describe the features (variable and common) of a product family in an unambiguous and manageable way. There are applicable methods but most commercial tools do not provide appropriate modeling notations, management support for design and evolution of the feature graphs. Several notations have been introduced [11][8][9], but notation-driven tool

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR),
Tampere, Finland, 24-26 March, 2004, pp. 261-269.

support is almost non-existent, so that design and updates of feature graphs is not supported.
Secondly, an appropriate set of features for a product should be retrievable from the family feature graph. This means an ability to select the set of appropriate features from the family feature graph, reorganize them and, in most cases, add new features that are needed for a particular product and might also be imported to the family feature graph. However, the new features should be added to the product without changes to the features already taken from the family feature graph; otherwise, the product does not conform to the family feature graph anymore.
Thirdly, product variants may have different architectural styles, e.g. two or three tier client-server architecture or the peer-to-peer architecture style for deployment. Therefore, architectural styles are selected for a single product and the selected features should be mapped on components and on the role these components play in the architectural style.
Fourthly, the selected set of features is mapped to components that realize the required functionality and quality properties. There are concept models and clustering techniques appropriate to organize features to groups [16]. A concept model describes semantic relationships between domain terms. Features are atomic terms that can be clustered within a concept or shared by different concepts. Clustering reduces the number of combinations and guides the component development. However, not all features can be mapped directly to components. Some of them are cross-cutting features that disperse to several components, interfaces and classes, data and methods inside a component. The tracing of cross-cutting features is not possible manually, and existing tools do not provide a solution for that.
Lastly, an increasing number of third party components is used in product families and, therefore, there is a need to add the features of new components to the product features. The assumption is that, in practice, a new off-the-shelf or commercial-off-the-shelf (COTS) component is first evaluated by using it in a single product and after checking the compliance of the component with other product features, it is added in the family feature graph. From the business point of view, the manageable use of third party components can improve multiple factors, which influences the achievement of fast, efficient, predictable, low-cost, high-quality production and maintenance. Examples of these factors include: the ability to take advantage of new products and new technology faster; the significant decrease of time-to-market because off-the-shelf components and COTS components are ready to use; higher employee productivity, with the emphasis not on coding but on (re)using and integrating.
In order to provide a tool that supports all aspects mentioned above, a new tool can be constructed or an existing one has to be adopted and extended with the required capabilities. Our approach is the latter, which is more practical in industrial settings. However, there are some requirements for the tool that is used to trace dependencies between features, architectural patterns, components and source code. The tool must be adaptable and open; existing properties of the tool have to be changed and new ones are to be added. The tool we selected for our experiment was Together® ControlCenter™, which provides extension mechanisms such as configuration files and integration of Java implementation modules. Starting with Together® ControlCenter™, we developed a tool that supports top-down as well as bottom-up traceability, from the product family feature graph all the way down to implementation files.

2 Product Family Representation

The representation of a family of software products has been organized on three abstraction levels (see Figure 1). The product family (PF) is modeled at the *PF Level*, by means of a PF Feature Map[1] (PF FM). Decisions taken at design time are captured by this FM, which includes all the features (both internal and acquired from third parties) and the variation points in the application domain.
Individual family products are represented at the *Product level*. Each product is described in terms of the subset of features it supports. The Product FM captures these features. This set of features is then translated into the design decisions captured by the Product Component Map (CM). With the separation of the Product Level from the PF level, we can make explicit the decisions taken for individual family members at deployment/configuration time, or at runtime.
Finally, the Product Level is on top of the *Implementation Level*, which represents the set of reusable implementation assets belonging to the product family. It includes, for each product, the implementation of each concrete feature and any associated documentation (e.g. user manuals, test material).

[1] Term *map* is defined as the representation of the whole or a part of an area. Representing features the application domain (or part of it), we use term *feature map* instead of *feature graph* to underline its role in providing traceability, navigation support and domain coverage.

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR),
Tampere, Finland, 24-26 March, 2004, pp. 261-269.

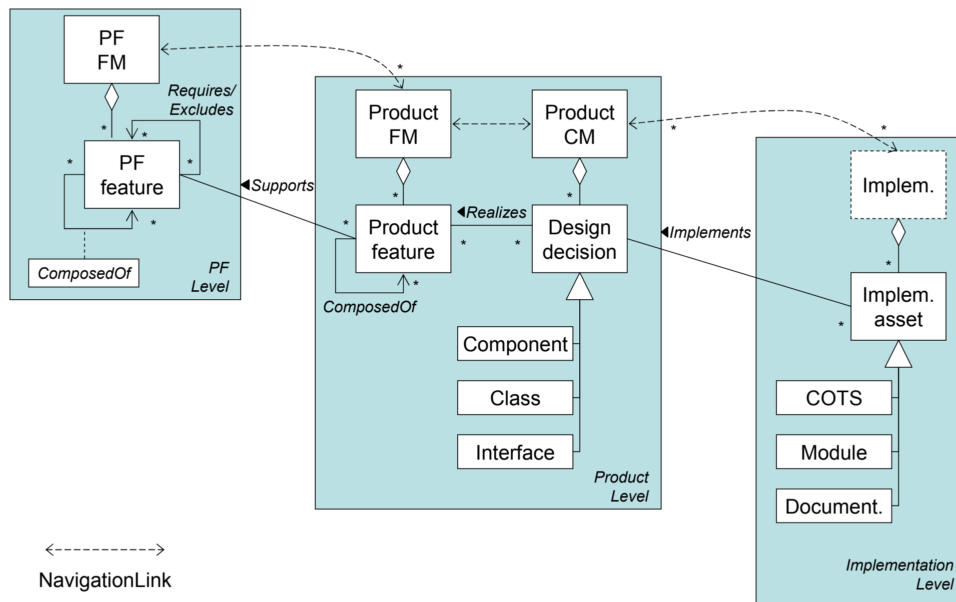

Figure 1. *Simplified representation model*

In our approach, top-down and bottom-up traceability is possible by following the relationships across abstraction levels and product-related maps:

- *Traceability between PF and Product features.* Traceability is possible by following the *NavigationLink* between FMs (the PF FM and the Product FMs). The *NavigationLink* is provided by association *Supports* defined between PF features and the Product features that are fulfilled by some product. Figure 1 also depicts the associations internal to the abstraction levels, and that are used to browse inside a FM. For instance, the PF features are organized in a decomposition hierarchy provided by association *ComposedOf* (also specifying the type of variability), and dependencies across features are modeled by associations *Requires* and *Excludes*. At the Product Level, the features are organized in a decomposition hierarchy in which variability has been solved through variants selection.
- *Traceability between Product FM and CM.* A Product FM is associated with the Product CM realizing it (see *NavigationLink* between Product FM and CM). The *NavigationLink* is provided by association *Realizes* defined between each Product feature and the design decisions captured by the Product CM and that can be components, classes or interfaces.
- *Traceability between Product CM and implementation.* Elements in a Product CM are associated with their implementation (see the *NavigationLink* between Product CM and Implementation). The *NavigationLink* is provided by association Implements defined between each design decision and the implementation assets solving it (e.g. executables like COTS components, source code modules, and associated documentation files).

The tool and underlying approach has been applied to a family of products in the next generation networks (NGNs) service domain. NGNs integrate hybrid telecommunication networks (like fixed telephony, packet switched and wireless networks) via middleware platforms, which hide network details and expose basic communication services to applications. In this way, applications can realize communication and multimedia services (examples are wired-wireless gaming, e-learning, virtual office environments, GIS, etc.) independently from the underlying network technologies, so that ubiquitous communication is achieved in an easy way.

This domain is particularly interesting as NGN service components implement a complex network of interactions, which involve many different technologies. Further, each product integrates hybrid technologies usually provided by third parties or

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR),
Tampere, Finland, 24-26 March, 2004, pp. 261-269.

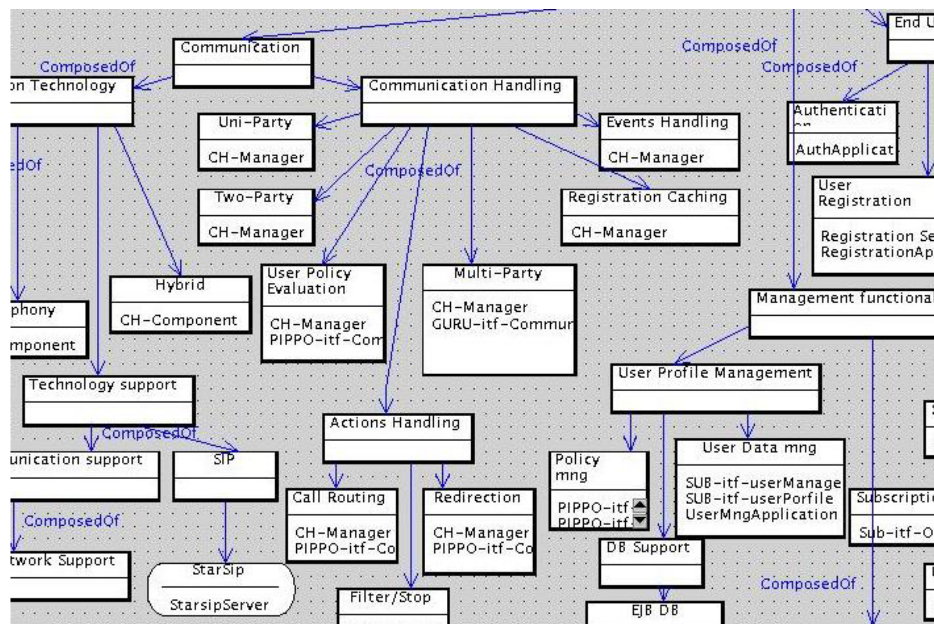

Figure 2. *A Product FM fragment with linked CM variants*

acquired as open source or COTS components. Hence, features are either reused or internally developed.

Our product family [13] consists of six products that have been recovered by following a bottom-up approach. Next, a seventh product has been derived following a top-down approach.

As an example, Figure 2 shows a fragment of a Product FM at the Product Level. It shows the features supported by a family member: each feature is visually represented by a white box organized in two areas showing the name of the feature, and the list of variants providing a solution to that feature. The variants are organized in a Component Map (depicted in Figure 3) at the Product Level. Traceability is made explicit by the link between each Product FM feature and its variants in the CM.

For example, in Figure 2 feature *Communication Handling* is decomposed into a list of refined features including *Multi-Party* and *User Policy Evaluation*. Feature *Multi-Party* is realized by two variants, object *CH-Manager* and interface *GURU-itf-Communication* (exported by component GURU). Feature *User Policy Evaluation* is realized by object *CH-Manager* too, and by interface *PIPPO-itf-Communication* (exported by component *PIPPO*). This means that object *CH-Manager* supplies two features in communication handling (namely communication between multiple parties, and evaluation of user policies).

In the tool, it is possible to browse the link to each variant associated to a feature: by selecting one of the variants, the user is led to the Product CM, and the mouse focus is set to the CM element (object, interface or component) modeling that variant.

Figure 3 shows the CM associated to the Product FM above: for instance, from feature *Multi-Party*, we can navigate the variant *CH-Manager* that leads in the CM to the associated object inside component CH (right hand side of Figure 3). In the same way we can navigate the variant *GURU-itf-Communication* that leads in the CM to the associated interface in component GURU (right upper corner of Figure 3).

In summary, traceability from features to variants is achieved by linking features in FMs with the variants in the CMs. In a similar way, traceability towards implementation is achieved by linking elements in the Product CM with their implementation assets (e.g. source code files, configuration files, interface descriptions, etc.).

3 Related work

Much seminal work in the area of traceability concerns requirements traceability, especially the traceability between requirements and later products such as designs and implementations [7]. Supporting traceability in product families is a relatively new area of research. Current approaches to traceability in product families generally define a model representing

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR),
Tampere, Finland, 24-26 March, 2004, pp. 261-269.

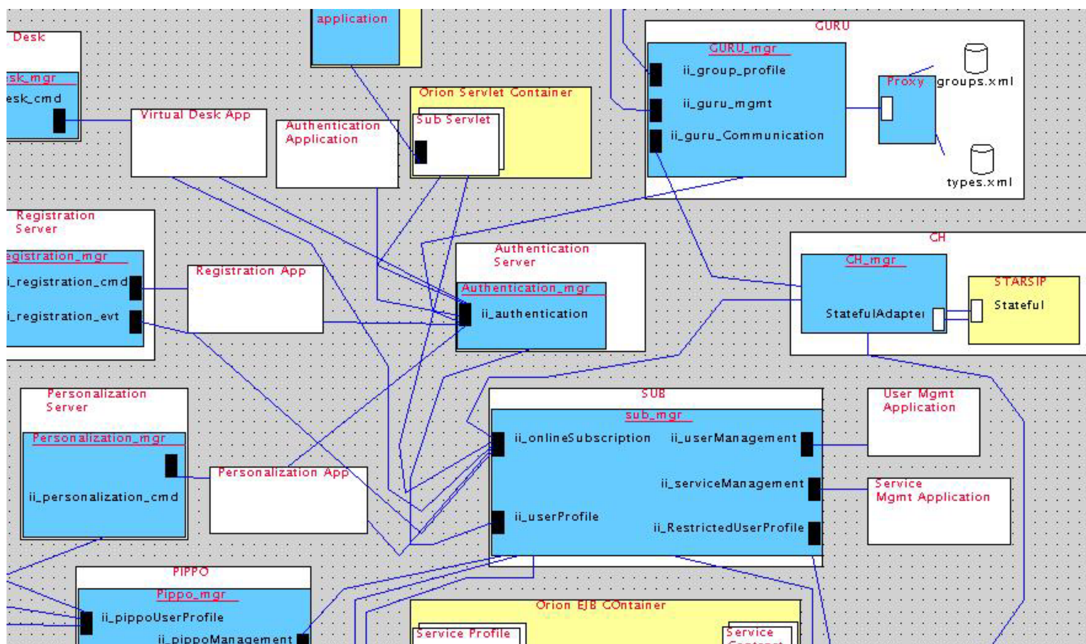

Figure 3. *A Product CM fragment*

traces, and provide an articulated coverage of both product and organizational issues. These mostly do not cover any tool support. In contrast, we do focus on tool support, but limit ourselves to product issues only. This restriction helps us in verifying the usefulness of our model and in identifying the needs for additional aspects.

For example, [14] defines an approach to model-based requirements engineering in which a product family PF and its products are represented in a document model, a requirements model, and a system model. Model elements (e.g. product requirements or functional decomposition of architecture) can be linked together by using a link "causes" that supports tracing the decomposition of requirements, and a link "derived" that allows tracing which PF requirements are fulfilled by which products. This work is in some sense broader than ours, in that it provides a very detailed description of the models and sub-models representing a product family and its products, as well as organization-related issues like support for parallel working groups. It also defines the link "is in version" associating a requirement with the product versions it is valid for. Rather, we focus on two simple models, the FM and the CM, and add a pragmatic definition of the complex net of links among these models' elements, and with their implementation assets. The main advantage is that our model has been complemented with tool support, which allows us to carry out experiments to reveal whether the model is good enough. Also, we do not consider the versioning problem, which adds a time dimension to the problem of representing variability (i.e. variability in time), but which can also be solved by integrating the tool with some configuration management product (further discussed in Section 5).

In the context of the SPLIT/Cloud method, [2] defines an approach to trace requirements. The traceability problem is split in traceability between functional and non-functional requirements, and traceability for derivation. The first is supported by a link "is realized by" (expressing that a non-functional requirement is realized by a functional requirement) and a link "is applied on" (connecting a functional requirement with the non-functional requirements it is constrained by). Traceability for derivation is supported by a link "excluded" (expressing alternative decisions) and "included" (expressing included functional requirements). In our work we focus on traceability for derivation. Non-functional requirements are not explicitly discussed, even though they can be covered by a refinement of the FM: in this case, traceability between functional and non-functional requirements can be achieved by links *Requires* and *Excludes*.

In [15] Ramesh et al. define a conceptual framework to represent traceability among various information objects. This framework basically relates requirements with design objects by making explicit the issues faced and the decisions taken during development. Here

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR), Tampere, Finland, 24-26 March, 2004, pp. 261-269.

links like “leads to” and “modifies” are defined between requirements and decisions, whereas links like e.g. “implies” and “creates” exist between decisions and design objects. This work provides an interesting solution to the problem of managing decisions/assumptions, which again is complementary to our work, as we focus on the relations between features, design solutions and implementation assets.
Finally, [1] adds traceability support to the PuLSE method by defining a two step method: first, a general meta-model defines potential traces between model elements; second, the meta-model is specialized for a specific project, to express the project-specific types of traces. The method could be used to generate our three-level model.

4 Discussion

4.1 Technical Issues

As introduced in Section 1, in order to provide proper support to traceability and product derivation in product families, there are several technical issues to be considered.

First, in the extension applied to the software tool, we added two types of diagrams, the Feature Map and the Component Map. Features at the PF Level and features at the Product Level are represented in terms of FMs, whereas solutions provided by family members are represented in CMs. Also, we defined the whole chain of links between PF FM, Product FM, Product CM, and Implementation. This link chain supports traceability in a smooth way, a requirement put forth in [19]. In the example, e.g. we showed that from feature Multi-Party (in Figure 2) we can identify the list of variants and, by navigating them we can see the associated CM (in Figure 3). This navigation is also possible from PF features to Product features, e.g. to identify the list of products that support a selected feature. Furthermore, management of features is eased by generating views focusing on the subset of features/dependencies that are needed e.g. to add new features to the product family.

An additional extension supporting traceability is the possibility to configure the link of Product CMs with their implementation: the physical location of the implementation assets can be changed without influencing the links with CMs and FMs. This extension is very useful in practice: a product family is a long lasting asset, and the storage place of its implementation is likely to change. Therefore, it must be possible to easily maintain the chain of links down to implementation.

Secondly, when we want to derive a new family member, we need tool support to focus on a selected subset of features, because during product derivation it is often difficult to identify from the many features in a domain, the subset of features we would like to include in a new product. To support the architect in this activity, it is possible to define a view focused on (1) a defined subset of features and (2) a defined list of relationships between features. In this way, the architect can easily find out the existing variants for the desired features, and their existing dependencies. Also, when a view is generated, traceability downwards to CM and implementation assets is maintained in our implementation. We believe that view generation and customization provides a powerful means to manage complexity.

Thirdly, the separation between PF and Product levels allows to assign to each individual product a particular architectural style, which can be used e.g. to drive product deployment. This assignment is supported in our tool extension, by associating a CM with a certain architectural style (defining component/connector types), and then by defining in the properties of each component the role played in the architectural style.

Fourthly, we need a smooth representation of the features supplied by a family member, the associated design solution, as well as their implementation. The introduction of the Product CM provides this necessary bridge between the feature representation of family members and their reusable implementation assets. Also, the CM describes the product design (in the solution space) associated to the product features (in the problem space). The maintenance of the relationships across the three abstraction levels (PF-Product-Implementation) offers real support for product derivation following a top-down approach. This can be done by selecting (in the PF FM) the features we want to use in a new product, and by browsing (in the Product FMs and CMs) the available solutions provided by various family members, to judge if they are suitable to new requirements. Suitability can be decided at the design level by navigating the Product CM and at implementation level by browsing the implementation artifacts along with the associated documentation.

In our experiment, we also carried out bottom-up recovery of six existing products. This has been done by extracting the Product CM from the design of each product, and by building a Product FM covering all the features provided by all CM elements. FM definition underwent various iterations, to harmonize feature decomposition and feature names among different product FMs. At last the PF FM has been defined to

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR), Tampere, Finland, 24-26 March, 2004, pp. 261-269.

include the features of the complete product family. This bottom-up feature definition was done manually, and when feature maps were completed, the links between PF FM, Product FMs and Product CMs were added. Automatic support for bottom-up recovery is still missing. Also, documentation (description, binding-time, taxonomy, etc.) of maps and map elements (e.g. features, components) has been added to the tool. A possible future extension includes the tool-supported generation of such documentation information.

Lastly, an important aspect in industrial settings is the possibility of explicitly representing third party components in terms of the features they provide, and their role in specific product solutions. This aspect is supported at both PF Level and Product Level. In the FMs we can represent features acquired from external sources in terms of *external features* – whose definition was already provided in [9][2]. In the CMs, the components providing these features are modeled as black box elements, and integrated with other components at the interface level. For example, the product modeled in Figure 2 includes SIP communication in the supported communication technologies (modeled as feature *SIP* in the left lower side of Figure 2). This feature is offered by a commercial SIP server (StarSIP™ by Telecom Italia Lab) that offers a set of APIs used to implement the *CH* component, as described in Figure 3.

4.2 Experience report

Validation: The tool has been validated in two steps. First a family of six related products in the Next Generation Network service domain has been recovered. The objective was to observe if notational and methodological support was sufficient to grasp all information needed to model the existing family. In this perspective validation was successful: we first defined a service classification based on service categories and properties; then we translated it into a PF FM so that the features supplied by the family products could be mapped on the domain features modeled in the PF FM. More precisely we may say that indirectly also our service classification proved to be suitable for domain modeling. All family products could be modeled: the notation is expressive enough to include all existing elements and dependencies at both the feature and component levels; also, all documentation belonging to the different products could be either translated into maps or associated with diagrams or diagrammatic elements belonging to the representation model of Figure 1.

Afterwards, when the product family was inserted in our tool, a seventh product from this family has been derived. In this second step we aimed at validating the support provided by the tool and by the method in communicating the knowledge represented. Even though validation objectives were rather informal, we can report some important lessons learned and issues needing further work:

- Traceability support is a fundamental factor in successful **knowledge communication** about reusable assets.
- Mechanisms to govern **complexity** are needed at all abstraction levels: in feature maps we need filtering and viewpoint generation to focus on the features and the dependencies we are investigating. Modularization of views needs careful investigation to be able e.g. to maintain their internal and external consistence.
- In product derivation, we desperately need some automated support for **initial composition** of reusable assets (at both the feature level and the structural component level). When we identify a list of desired features that we want to have in a new product, we also want to know if these features are compatible, and/or which are the constraints that we inherit by composing them. Further, when we choose for each of them a certain design/implementation solution, we have to answer the same question again. Dependencies provide a first insight in this respect: e.g. we can identify whether two components rely on incompatible programming languages, or whether their interfaces are inconsistent. Nonetheless, richer semantics needs to be associated with such dependencies. Further investigation is needed to state more formally how dependencies must be specified and how tool support can help in this respect.

Scalability: The product family used in our experiment could be claimed to be small (seven products, 20,000 SLOC/product). On the other hand, it demonstrated to be complex enough (especially in terms of types and number of cross-dependencies) to raise some interesting research questions. Industrial product families count thousands of products, each made of several millions SLOC. Of course we have not demonstrated that our approach is scalable to these numbers. Nonetheless, the lessons learned discussed

---

[2] Our definition of *external feature* is broader than the one given by Gurp et al. [9]. They consider an external feature as provided by the platform (i.e. external environment). In our case, we can cover platform features too, by considering them as black box entities from the perspective of the current development.

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR), Tampere, Finland, 24-26 March, 2004, pp. 261-269.

above provide a first step in this direction. The state of the practice in managing large industrial product families often relies on informal models and inadequate tool support; the natural next step is to investigate how our approach together with tool support can improve the state of the practice.

Unambiguous representation of features: An important aspect in traceability is the ability to identify a desired feature and its possible solutions in an unambiguous way. This is naturally supported by our approach: by separating the PF level and the Product level, we maintain a generic feature model of the domain (in the problem space) and an explicit feature map for each product (in the solution space). Moreover, each feature at the PF level is associated with all the products that provide a solution to that feature, so that we can identify any solution unambiguously.

Generation of traceability information: In product derivation it is customary to start by selecting the desired features that we want to include in the new product, and then add new features or modify existing solutions to already supported features. This gives raise to the problem of (1) including (or not) the new features in the PF FM (cf. the problem of augmenting the borders of the product family) and (2) updating the PF FM to trace the solutions devised in the new product. When we include existing features, tool support automatically includes all traces associated with such features. In particular, when we reuse one of the available feature solutions (modeled by association *Supports* between the PF level and the Product level in Figure 1), we select the chosen Product feature that we want to reuse, and we automatically get all the traces to its design (from the Product CM) and implementation. When instead we add new features or modify an existing solution, we currently need to manually define/update all traces. The complexity of automatically supporting evolving features is that (1) to include a new feature in the PF FM we need to decide the feature category and its position in the FM; (2) to update the PF FM to trace product-specific feature solutions is just an implementation matter; instead, to trace the role played by the new feature in the associated Product CM, is again a semantic problem. E.g. we need to know whether a feature is implemented by a new component or by an interface only. In this context, tool support can provide guidance to drive traceability maintenance, by identifying inconsistencies and highlighting them to the stakeholder.

Architectural styles and patterns: As discussed in Section 4, the tool supports the application of styles and patterns by assigning component/connector types and component roles. Nonetheless more advanced support is needed. For example, the ability to generate one (or multiple) structural views according to the chosen architectural style(s), or the static verification of the properties envisaged by the style(s) or pattern(s) would provide much help during product design. The Product CM is the natural base on which such tool support can be constructed: as CMs are at the Product level, they can help tracing the decisions about the architectural styles and patterns adopted by the product.

5 Conclusions and Future Work

We defined a product-oriented model to represent features and design decisions as well as implementation assets at both the PF Level and the Product Level. Our model supports cross-level traceability, and has been used to extend a commercial software tool, Together® ControlCenter™, with the following capabilities:

- It allows us to describe features of a product family in an unambiguous and manageable way.
- It allows us to select a set of appropriate features, reorganize them and add new features to develop a specific product.
- It allows us to apply different architectural styles to different product variants.
- It allows us to trace cross-cutting features.
- It allows us to model both newly developed and COTS components.

The tool has been validated by recovering the family of six related products in the Next Generation Network service domain, and next deriving a seventh product from this family. Though related work enabling traceability exists, our model especially addresses the problem of representing product-specific information, and provides a pragmatic solution implemented by tool support.

Future work includes some shortcomings of the tool: it does not yet provide support to generate code, or to generate a (tentative) product component map by combining variants of different products (i.e. of different product component maps). Also, we will consider porting our implementation on the Eclipse open platform [4].

Ongoing work is studying how to integrate the tool with Ménage, an environment aimed at providing versioning and configuration management for product families. As Ménage focuses on design and implementation and it does not support the early development phases, this integration will guide the full life cycle.

In: Proceedings of the Eight European Conference on Software Maintenance and Reengineering (CSMR), Tampere, Finland, 24-26 March, 2004, pp. 261-269.

An important research issue in traceability concerns the representation of assumptions. In developing a new product we decide for a certain design solution (e.g. client-server instead of three-tiered) or for a list of quality characteristics (e.g. portability and modularity). In doing this, we implicitly assume that certain conditions will hold. For example, we suppose that the product will be modified to accommodate a certain (expected) evolution, and that the product will be designed to reflect the structure of the development team. In summary, we implicitly make assumptions that impact our solution but that we do not formalize in any documentation (i.e. that remain *implicit*). The drawback is that when reusing a solution we also reuse its implicit assumptions, and when we apply modifications we might introduce conflicts that will be discovered later on.

The natural next research step is to investigate how implicit assumptions can augment the PF architectural knowledge, and how they can be traced to support reuse better.

References

[1] J. Bayer, T. Widen, "Introducing Traceability to Product Lines", Proceedings of the fourth Workshop on Product Family Engineering (PFE-4), LNCS 2290, F. van der Linden (Ed.), Springer Verlag, 2002, pp. 409-416.

[2] M. Coriat, J. Jourdan, F. Boisbourdin, "The SPLIT Method", Proceedings of the First Software Product Lines Conference (SPLC1), Denver, Colorado, USA, Aug. 28-31, 2000, pp. 147-166.

[3] L. Dobrica, E. Niemelä, "Using UML Notation Extensions to Model Variability in Product-line Architecture", International Workshop on Software Variability Management, ICSE'03, Portland, Oregon, May 3-11, 2003, pp. 8-13.

[4] The Eclipse open platform, on-line at http://eclipse.org.

[5] A. Garg, M. Critchlow, P. Chen, C. Van der Westhuizen, and A. van der Hoek. "An Environment for Managing Evolving Product Line Architectures", International Conference on Software Maintenance, Sep. 2003, pp.358-367.

[6] H. Gomaa, M. E. Shin "Multiple-view meta-modeling of software product lines", The eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Maryland, USA, Dec. 2002.

[7] O.C.Z. Gotel, A.C.W. Finkelstein, "An Analysis of the Requirements Traceability Problem", First International Conference on Requirements Engineering, IEEE, 1994.

[8] M.L. Griss, J. Favaro, M. d'Alessandro, "Integrating Feature Modeling with the RSEB", Fifth International Conference on Software Reuse. Los Alamitos: IEEE Computer Society, 1998, pp. 76-85.

[9] J. van Gurp, J. Bosch and M. Svahnberg, "On the notion of variability in software product lines", Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, 2001, pp. 45-54.

[10] J. Kalaoja, E. Niemelä, H. Perunka, "Feature modeling of component-based embedded software", Proceeding of 8th IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering, Los Alamitos, USA: IEEE Computer Society, 1997, pp. 444-447.

[11] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A. S. Peterson, "Feature-oriented Domain Analysis (FODA)", Feasibility Study, Technical Report CMU/Sei-90-TR-21. Software Engineering Institute. Pittsburgh, USA: CMU, 1990, 147 p.

[12] P. Knauber, S. Thiel, "Session Report on Product Issues in Product Family Engineering", Proceedings of the fourth Workshop on Product Family Engineering (PFE-4), LNCS 2290, F. van der Linden (Ed.), Springer Verlag, 2002, pp. 3-12.

[13] P. Lago, "A Policy-based Approach to Personalization of Communication over Converged Networks", IEEE International Workshop on Policies for Distributed Systems and Networks, Monterey, CA, USA, Jun. 2002.

[14] J. Plankl, G. Böckle, "Modeling Concepts for Product Families", Requirements Modeling and Traceability, ESAPS report, Nr. Philips-WP3-0106-01, 2001.

[15] B. Ramesh, A. Tiwana, K. Mohan, "Supporting Information Product and Service Families with Traceability", Proceedings of the fourth Workshop on Product Family Engineering (PFE-4), LNCS 2290, F. van der Linden (Ed.), Springer Verlag, 2002, pp. 353-363.

[16] M. Simos, *Organization Domain Modelling (ODM),* Guide book, Version 1.0, STARS-VC-A023/011/00. March 1995.

[17] M. Simos, J. Anthony, "Weaving the Model Web: A Multi-Modelling Approach to Concepts and Features in Domain Engineering", Fifth International conference on software reuse, Los Alamitos, USA: IEEE Computer Society, 1998, pp. 94-102.

[18] D. Webber, H. Gomaa, "Modeling variability with the variation point model", Proceedings of the International Conference on Software Reuse, LNCS 2319, C. Gacek (Ed.), Springer Verlag, 2002, pp. 109-122.

[19] T. Weiler, "Modeling Architectural Variability for Software Product Lines", Workshop on Software Variability Management, Gröningen, The Netherlands, Feb. 2003, pp. 55-63.