Version management tools as a basis for integrating Product Derivation and Software Product Families

Jilles van Gurp, Christian Prehofer
Nokia Research Center, Software and Application Technology Lab
{jilles.vangurp|christian.prehofer}@nokia.com

Abstract

This paper considers tool support for variability management, with a particular focus on product derivation, as common in industrial software development. We show that tools for software product lines and product derivation have quite different approaches and data models. We argue that combining both can be very valuable for integration and consistency of data. In our approach, we illustrate how product derivation and variability management can be based on existing version management tools.

1. Introduction

The last ten years have seen an increasing interest in software product lines [1]. This has started from different approaches such as generative programming [2], feature-oriented design [4] and programming [6]. By now, there is a large amount of research on software product lines and variability management, which has led to several tools and various industrial experience, e.g. for mobile phones [3].

This paper considers tool support for variability management, with a particular focus on product derivation, as common in industrial software development. The purpose of product derivation in a product family is to construct a software product from a base platform consisting of architecture, design and reusable code. The product derivation process consists of selecting, pruning, extending and sometimes even modifying the product family assets. Additionally, in many companies which practice product family development, this is not a one time activity but a process that has an iterative nature:

- Usually, both product family and derived products evolve independently. They each have their own roadmaps and deadlines. However, it may be desirable to propagate changes from the product family to already derived products (e.g. bug fixes or new features) at certain points in time.
- As outlined in [9], repeated iterations are often required as understanding of the product requirements progresses.

We discuss in the following tool support for software product lines and product variation. For instance, common tools for product derivation are version management systems, such as Subversion [11][12]. We show that both areas have quite different approaches and data models. We argue that combining both can be very valuable for integration and consistency of data. In our approach, we sketch how product derivation and variability management can be based on existing version management tools.

1.1 Tool support and product derivation

To support product derivation, various research papers have proposed using variability models and tools. Some of these approaches have been applied successfully in practice as well. The problem of describing and representing variability in these models is well covered in the research field. Essentially the tool support for this can be broken down into two categories (though some tools arguably fit in both categories):

- **Build oriented tools.** Tools that integrate into the build process. Examples of such tools include KOALA [10] and PROTEUS [7]. These tools enable the selection and configuration of components and generating glue code.
- **Documentation oriented tools.** Tools that focus on documenting the provided variability and that provide traceability of requirements and variation points to code. A good example of this is e.g. COVAMOF [9] and VARMOD [8]. These tools are primarily used to guide the (manual) process of product derivation.

Both categories of tools are very useful in their own right. Most of the above research has mostly been centered on the requirements and variability aspects. However, the process of product derivation (i.e. exploiting the provided variability in the software

product family to create product variants) is only partially supported by tools.

Product derivation is about more than this. It includes:

- Selecting components. Reusing components provided by the software product family.
- Overriding components. Replacing provided components with alternative implementations (provided by the software product family or product specific).
- Modifying provided components. Sometimes product requirements conflict with product family requirements. Adapting such code in the derived product is a solution that despite its disadvantages is preferred in many companies.
- Providing new variants for existing variation points (e.g. implementing component interfaces)
- Adding product specific components and architecture.
- Configuring reused, modified and product specific components.

We observe that none of the variability management tools fully supports all of these activities. Secondly, we observe that the product derivation process, like the rest of the development process, is iterative. In other words, it is not a one time activity but a recurring activity during the evolution of a product.

2. Supporting product derivation with version management

The solution we propose involves exploiting functionality provided in common version management tools. The advantages of doing this are:

- Version management tools are used anyway in development organizations so it is a relatively easy transition for development teams to start using these tools for product derivation as well.
- Version management tools are the place to keep track of relations between artifacts (typically components) both in space (branches) as in time (revisions). Derived components can be seen as product specific branches of product family component.

Notice that version management works on any development artifacts (directory, a file, or an entire subsystem). We identify files or directories with components for simplicity.

In version management terms, product derivation is equivalent to creating a branch (or branches) of the main product platform and then committing product specific changes on these branches. However, this is not how version management tools are currently used in many product family using organizations. Instead products are usually created by copying (or generating configurations) artifacts from the product family and then adding product specific artifacts. This is especially problematic when product specific changes need to be made to the copied artifacts.

This is very similar to the notion of having multiple branches of the same code base in a version repository. A version management system supports this type of functionality by:

- Keeping track of the changes
- Allowing for changes to be merged from one branch to another

2.1 Subversion

There are roughly three generations of version management systems:

- Individual file based systems like RCS. Manage individual files. These systems are rarely used these days.
- Systems that can version groups of files (CVS and many commercial version management systems). While still popular, these systems lack many of the features that would enable using them for product derivation.
- Change-set oriented version management systems. These include systems like Subversion and GIT. The key difference is that rather than versioning individual files (like CVS) changes to the complete system are versioned with all its aspects including file system manipulation, symbolic link creation, meta-information modification and file changes etc. The delta between two revisions of the system in the version management system is called a change set.

Subversion [11][12] is a good example of a third-generation version management system. For the remainder of this paper we will assume Subversion or similarly capable, change set oriented version management system when we refer to version management. Our approach requires many of the features common in this new generation of version management tools. A few essential features are:

- **File system based rather than file based.** It can version all file system activities, including deletion, moving, linking and copying. For example, the history of a renamed file includes all commits before it was renamed; the rename; and all commits after it was renamed.
- **Copy by reference.** Copies are always by reference. The consequence of this is that a copy preserves version history and that making copies in the version repository is both fast and cheap in

using server-side storage (unlike CVS where version history is not preserved and copying actually results in a full copy by value on the server).

- **Flexible repository layout.** Branching and tags are implemented as copies (by reference). Unlike many second generation versioning systems, branching and tagging are not special operations. For example, creating a new branch from trunk amounts to making a copy of a specific revision of the trunk directory to the branches directory. Subversion repositories (by convention, not by rule) contain directories with the names trunk, branches and tags. However, whether these directories are located directly under the root or deeper in the directory structure is up to the repository maintainer. In fact, using the subversion move operation it is trivial to change the directory layout if needed.
- **Properties.** Subversion supports annotations by associating properties (name value pairs) with any artifacts under version management. Of course changes to properties are also properly versioned (so they property manipulation is part of the version history).

2.2 Information models

The information model used by most variability tools is very different from that used by version management tools such as subversion. In the context of product derivation, both models are relevant. Therefore, we provide a brief outline of both in this section.

As outlined in the introduction the purpose of most variability tooling is to support product derivation either by documenting the variability in the software and/or by automating parts of the derivation process (e.g. component configurations; build configurations). Most approaches are centered on requirements, features and development artifacts. The information used by such tools consists of:

1. Feature models. A common way to model variability is to construct feature diagrams with variant features. These models may be textual or graphical.
2. Mapping of features to requirements.
3. Mapping of variant features to design and implementation level artifacts. Especially the build oriented tools require this information in order to support the derivation process.

The information model of version management systems on the other hand is concerned with managing the changes of files and directories. The information it manages consists of

1. A tree of directories, files and associated meta data properties. Usually the tree structure is derived from the logical architecture. For example, each directory represents a particular subsystem or module.
2. In subversion, the meta data properties mentioned under 1 are used to represent various properties related to versioning (revision number; commit message; data and time) the file content (character used for new lines in text files; the mime-type of the file content; etc). Additionally files and directories may be annotated using custom properties. Subversion does not do anything with these properties (except for tracking changes to this data) but they may be used in scripts or custom applications that integrate the subversion programming API (bindings for C, python and Java exist).
3. Storing change sets between revisions of the versioned tree. In subversion, each commit to the version repository is stored internally as a delta to the previous revision of the repository (and unlike CVS, the revision always refers to the entire contents of the repository instead of artifacts in the repository).

2.3 Integrating both information models

As can be seen from the description above, both information models have different purposes. Although there may be little overlap in both, consistency can be an issue. Furthermore, product derivation support that goes beyond the regular branching and merging functionality, requires integration of these information models.

There are two strategies for doing this:

- Integrate version management information in existing variability tooling (e.g. by storing subversion URLs and revision numbers of relevant artifacts in the repository).
- Store variability model information and mappings into the version management repository.

The latter strategy may be supported using subversions annotation feature. Since version management systems store development artifacts, this concerns mostly storing the mapping of features and variability models to development artifacts. Using, for example, a "mandatory" property component directories corresponding to non optional features in the feature model could be marked as such. Similarly, a "depends-on" property might be used to indicate feature dependencies on other components in the repository. In this way, the information is directly stored with the corresponding code, which can help in

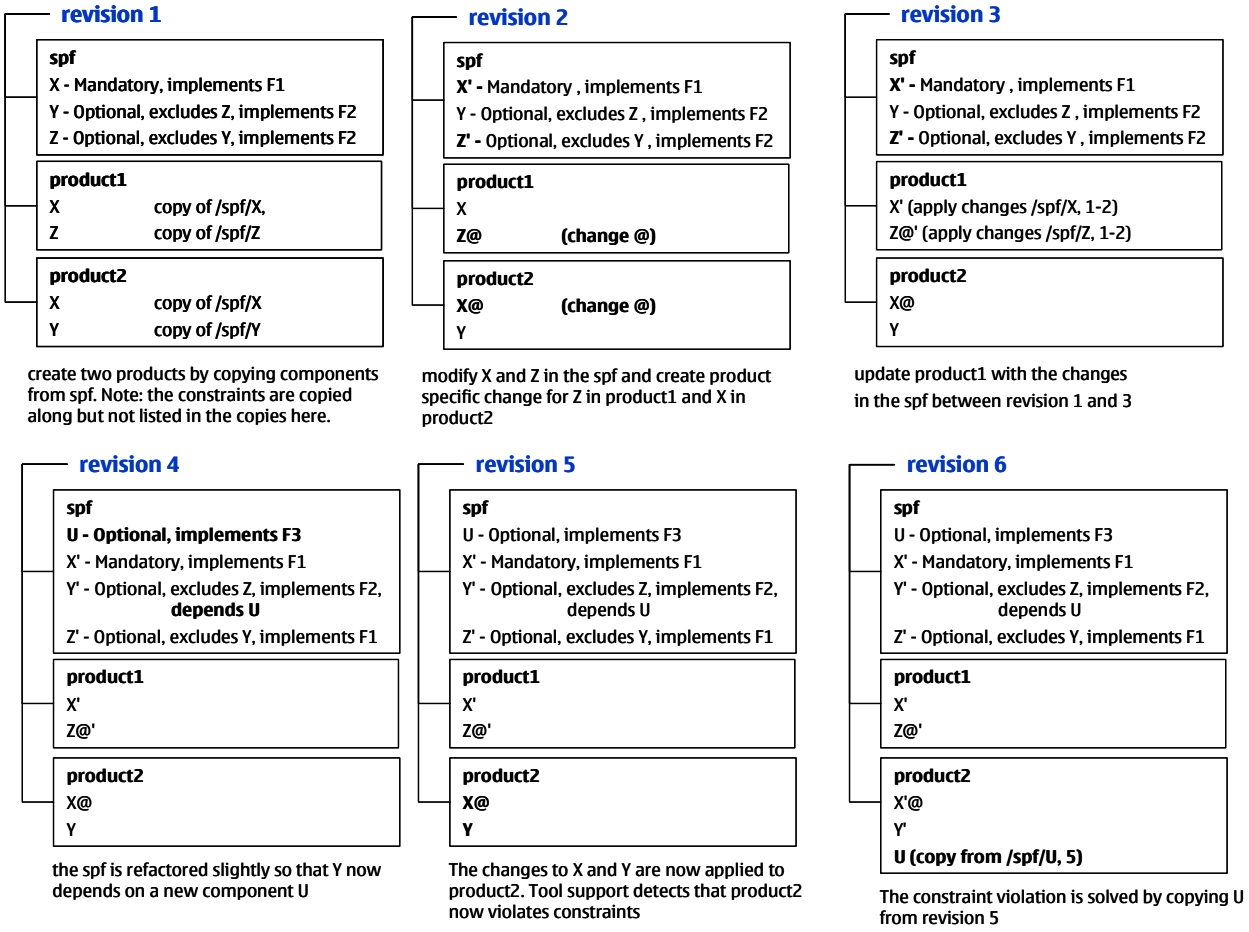

Figure 1: Example version management for product derivation and families

avoiding potential inconsistencies when working with different data bases.

2.4 Using the information model

To illustrate how product derivation would work with such an integrated information model, we run through a small example scenario based on an imaginary SPF that we follow through a few revisions, as shown in Figure 1 below.

In revision 1, an spf (software product family) directory is created and a few components are (X, Y and Z) are added. Using subversion properties it is specified that X is a mandatory component and implements feature F1. Similarly, Y and Z are optional and are variants of the same feature F2. Since a product can use only one of the implementations of F2, both implementations Y and Z exclude each other.

Furthermore, revision 1 includes two product derivations in the form of two directories in the repository (product1 and product2). Copies from spf have been made of X and Y for product1 and X and Z for product2. Althoug we do not show this explicitly, the properties on the spf components are copied as well. This allows us to use a tool to validate the constraints (in this case there are no violations).

In revision 2, we do some maintenance on the spf. This results in changes to /spf/X and /spf/Z. We indicate these changes using a ‘. Additionally product engineers make a product specific change to /product1/Z and /product2/X. These changes are indicated with a @.

In revision 3, product1 is updated with the changes made to the spf in revision 2. /product1/Z now has both the product specific changes and the spf changes. It might be possible that these changes are conflicting in which case the conflict would have to be resolved. It is worthwhile to point out that this conflict could have been identified (using a so-called dry-run for the merge of the changes on all the derived product components) already in revision 2 when the change was made to spf/Z. In a real product family, the ability to analyze the impact of important changes on derived products is of course a very important feature any potential conflicts might result in these changes to be

reconsidered or in some kind of upgrade strategy for the affected products.

In revision 4, some more refactoring is done on the spf. A component U is added and some changes to Y result in a dependency between Y and U.

In revision 5, product2, which is still based on the revision1 of the spf, is updated with the changes to X and Y. This results in a situation where constraints are violated.

Revision 6 resolves the constraint violation by copying U to product2.

The scenario above can be enhanced with tool support based on the information in the version repository. For example, constraints validation could be automated and run before each commit; as part of a nightly build or even integrated into the IDE. Similarly, impact analysis of changes on the SPF could be supported by trying to merge the changes to each of the derived products. These are just two simple but extremely useful ways to provide tool support using subversion.

3. Conclusions and future work

In this article we have outlined first ideas for complementing existing tools for product derivation based on software variability modeling with version management functionality in order to better support the derivation of software products. Our approach is especially appropriate in situations where it may be expected that:

- Derived products may include modifications to the components that they are derived from.
- Changes made to the product family after the initial derivation takes place need to be propagated to derived products.

The main purpose of this position paper is to shape our ideas with respect to future work:

- Provide a more formal definition of the information models.
- Explore additional opportunities for automating product derivation steps.
- Build layer of tools on top of subversion and existing variability tools and validate concepts using a case study.
- Explore advantages of using distributed version management systems where change sets are pulled rather than pushed, a notion that shifts control from product family developers to product developers.

References

[1] J Bosch, Design and use of software architectures: adopting and evolving a product-line approach, - 2000 - ACM Press/Addison-Wesley Publishing Co., New York, NY K.

[2] Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[3] Savolainen, J. Oliver, I. Mannion, M. Hailang Zuo, Transitioning from product line requirements to product line architecture, Computer Software and Applications Conference, COMPSAC 2005.

[4] Kang, C. K., Lee, J., Donohoe, P., Feature-Oriented Software product line Engineering, 2002, IEEE Software, 19, 4, 58-65.

[5] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In Object-Oriented and Internet-Based Technologies. 2004.

[6] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In ECOOP'97, 1997.

[7] E. Tryggeseth, B. Gulla, R. Conradi, Modelling Systems with Variability using the PROTEUS Configuration Language, Lecture Notes In Computer Science Vol. 1005, Springer-Verlag, pp 216 - 240, 1995.

[8] Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg 2005.

[9] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, Jan Bosch: Modeling Dependencies in Product Families with COVAMOF. ECBS 2006: 299-307.

[10] R. van Ommering, Building product populations with software components, proceedings of the 24rd International Conference on Software Engineering, pp. 255-265, 2002.

[11] C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick, Version Control with Subversion, O'Reilly Media, 2004, available at svnbook.red-bean.com

[12] The Subversion Project Home, http://subversion.tigris.org