Managing Forked Product Variants

Julia Rubin[1,2], Andrei Kirshin[2], Goetz Botterweck[3], Marsha Chechik[1]

[1]University of Toronto, Canada
[2]IBM Research at Haifa, Israel
[3]Lero, University of Limerick, Ireland

mjulia@il.ibm.com, kirshin@il.ibm.com, goetz.botterweck@lero.ie, chechik@cs.toronto.edu

ABSTRACT

We consider the problem of supporting effective code reuse as part of Software Product Line Engineering. Our approach is based on *code forking* – a practice commonly used in industry where new products are created by cloning the existing ones. We propose to maintain meta-information allowing organization to reason about the developed product line in terms of features rather than incremental code changes made in different forks and to detect inconsistencies in implementations of these features. In addition, we propose to detect and maintain semantic, implementation-level *require* relationships between features, supporting the developers when they copy features from different branches or delete features in their own branch, thus facilitating reuse of features between products. Our approach aims at mitigating the disadvantages of the forking mechanism while leveraging its advantages. We illustrate the approach on an example, and discuss its possible implementation and integration with Software Configuration Management systems.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reuse Models*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version Control*

General Terms

Design, Algorithms

Keywords

Software product lines, software configuration management, SCM.

1. INTRODUCTION

Software Product Line Engineering (SPLE) approaches support development of products from a common set of core assets in a prescribed way [3, 19]. These approaches advocate strategic, planned reuse that yields predictable results. However, in reality, product lines often emerge *ad-hoc*, when companies have to release a new product that is similar, yet not identical, to existing ones. To implement new product functionality, developers often *fork* an existing product and *modify* it to fit the new requirements (the "clone-and-own" approach) [17, 6].
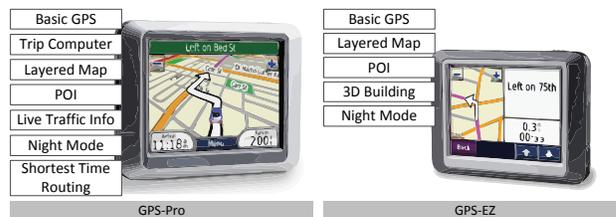
Figure 1: Two GPS products and their features.

Forking is typically perceived to be the easiest and the fastest reuse mechanism, providing the ability to start from existing already tested code, while having the freedom and independence to make necessary modifications to it. The disadvantages of this approach are also well-known: keeping track of changes made to each of the forked variants and propagating the changes between the variants increases the maintenance effort as the number of products grows. We illustrate some of these issues on the following example.

Example: GlobalCo is a company that delivers GPS solutions. It develops an advanced product (GPS-Pro) that has the Trip Computer feature for monitoring the vehicle speed and the time to destination, and the Layered Map feature for overlaying graphical objects on the map. The product is well tested and released to the market.

At some point, GlobalCo market analysis reveals the need for a simplified and less expensive variant of the product (GPS-EZ). Layered Map is determined to be the only essential advanced feature of this product, while Trip Computer should not be included. The easiest way to cope with this request is to fork the already tested code of GPS-Pro and remove the Trip Computer feature implementation from it.

Meanwhile, the development of GPS-Pro continues, and the ability to show points of interests (POI) and Live Traffic Info as additional layers on the map are added. It is decided to extend GPS-EZ with the POI feature but not with Live Traffic Info; thus the implementation of the POI feature should be copied from GPS-Pro.

In parallel, the development team of GPS-EZ implements the ability to show 3D Buildings as an additional layer on the map. The team also implements an extension to the POI feature but it cannot be immediately propagated to GPS-Pro because its current version of code is frozen towards a close release.

At this point, the GlobalCo marketing department decides to reassess the product portfolio. They ask the Product Line Manager which features are implemented in which products and whether there are differences in how these features are implemented. For example, they want to know that the POI feature works differently in GPS-Pro and GPS-EZ. However, since products are managed independently, there is no global view on the set of specific changes performed by each team.

Moreover, if GlobalCo chooses to add new features Night Mode and Shortest Time Routing to GPS-Pro, it is not clear whether they can also be easily incorporated into GPS-EZ: the Night Mode feature

is not designed to work with 3D Buildings, because GPS-Pro does not contain it. The Shortest Time Routing feature *requires* the Live Traffic Info feature that GPS-EZ does not have. Yet the information about such dependencies between features or even which feature exists in which product is not readily available.

To assist GlobalCo and to facilitate similar scenarios, the development environments should support the following tasks:

T1. Identify features implemented in the entire product portfolio and in each individual product.

T2. Discover inconsistencies between implementations of features in different products.

T3. Selectively copy feature implementations between products while resolving possible interaction conflicts – those that occur when the integration of two features would modify the behavior of one of them.

T4. Selectively remove feature implementations from a product.

The first two tasks support management of the product portfolio and decisions about its evolution, whereas the second two support feature exchange between products. These tasks rely on the ability to (1) *group* low-level changes into semantically meaningful units of functionality and (2) maintain implementation-level *semantic require dependencies* between these units.

Related work. Forked product variants are often managed by the Software Configuration Management (SCM) systems as distinct branches [28, 29]. Several authors aim to integrate SPLE concepts with SCM [27, 25], focusing mostly on deriving products from a common branch that encapsulates the entire product line and on managing the derived products. Some approaches capture composition constraints between different *versions* of software components that are stored in an SCM system and allow assembling a configuration containing just those component versions that satisfy the constraints [9, 31]. [4] proposes a heuristic for detecting artifacts that frequently changed together in different user activities. [16] describes a technique for *visualizing* dependencies between changes in an SCM system, but does not explicate what these dependencies are or how they are collected. [2] focuses on providing SCM capabilities that reflect the hierarchical structure of programming projects and teams that work on them. Numerous works suggest promoting team awareness by sharing information about changes and potential conflicts across branches [21, 10] or facilitating consistent editing of cloned code in a single branch [18]. The language-based approach of [22] represents a product line as a core module and a set of delta modules; user-defined ordering between these is intended to capture semantic dependencies between the features. Yet we are not aware of approaches aiming to organize existing, language-independent SCM-level changes into semantically meaningful group, i.e., features, and to detect implementation-level *require* dependencies between these features.

Contributions. Similarly to other authors, we believe that SPLE can rely on *forking* for realizing code reuse, if the approach is well-managed [24, 27, 25]. We thus propose to improve the efficiency of forking practices used in industrial organizations while mitigating their disadvantages. Specifically, we make the following contributions:

1. We define the *Product Line Changeset Dependency Model (PL-CDM)*, which captures the necessary information required for managing forked product variants. PL-CDM stores grouping of related code modifications into meaningful functionalities and grouping of those into user-level features. It also stores semantic *require* dependencies between the functionalities and the features, as well as the information on which functionalities and features are included in each product.

2. We suggest ways of building PL-CDM and discuss possible integration with existing SCM tools.

3. We demonstrate our approach for managing and evolving forked product variants on the GlobalCo example.
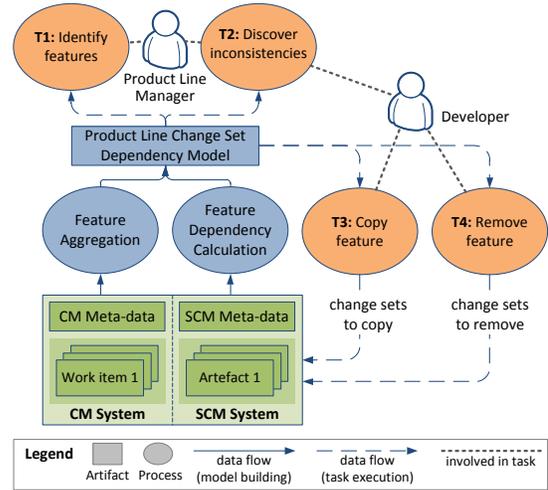
Figure 2: Overview of our approach.

The remainder of this paper is organized as follows. We provide an overview of the approach in Section 2. The PL-CDM model and its uses are described in detail in Section 3. In Section 4, we discuss ways of implementing our approach. Section 5 concludes the paper and outlines ideas for future work.

2. OUR APPROACH

In this section, we outline our approach, schematically illustrated in Figure 2. Its core part is the *Product Line Changeset Dependency Model (PL-CDM)*. It contains information about the entire product line: its products, features of these products, and relationships between the features.

PL-CDM is built by extracting and analyzing data contained in the SCM repository and possibly other sources of information such as a Change Management (CM) system. This data analysis has the following goals:

1. *Feature Aggregation*: grouping incremental code changes into higher level concepts of SPLE, such as features. This facilitates raising the level of abstraction and reasoning about the developed product line using the accepted SPLE terminology and management mechanisms.

2. *Feature Dependency Calculation*: detecting semantic dependencies between feature implementations, such as *require* dependencies indicating that one feature implementation cannot be realized without the other. This facilitates isolating the minimal functionality essential for a feature to operate.

The above goals are archived by the corresponding processes in Figure 2. These processes can leverage a number of techniques, including static code analysis, and analysis of CM and SCM metadata (e.g., information about branching) (see Section 4).

The information stored in PL-CDM is used for different SPLE-related tasks. For example, Product Line Managers can query PL-CDM to obtain the complete set of product line features and the products that contain them (task T1). They can also detect products that have inconsistent implementation of a feature, i.e., products containing different sets of functionalities for a given feature (task T2). Software Developers can also consult PL-CDM for detecting *require* relationships between features and their corresponding functionalities. These dependencies specify the entire set of functionalities that need to be copied (resp., removed) when a feature is copied (resp., removed) (tasks T3 and T4).

Various automatic tools can use PL-CDM as well. For example, the SCM system itself can use it to issue *notifications* to developers of one of the products when a feature they use was modified in another product of the same product line. These *active notifications*, at the granularity level of a feature, can help keep the developed products in sync and prevent late and painful synchronization.
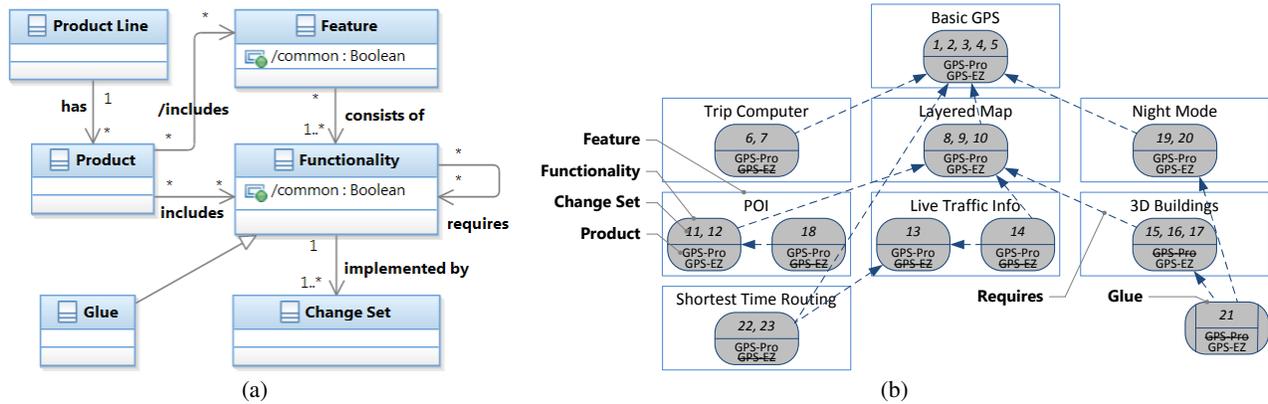
Figure 3: (a) the PL-CDM meta-model and (b) its instance, representing the GlobalCo GPS product line.

3. THE PRODUCT LINE CHANGESET DEPENDENCY MODEL

In this section, we describe the PL-CDM in more detail. We start by describing the meta-model underlying the information stored by PL-CDM (see Figure 3(a)) and then show how the captured information facilitates common product line management and evolution tasks.

3.1 The PL-CDM Meta-Model

Functionality is a central object of PL-CDM. It represents a software module that fulfills a specific purpose. Functionalities are implemented by a collection of individual *Change Sets* – a notion used by most modern SCM systems to represent sets of changes (additions, modifications and removals) delivered by a developer in a *single commit* operation. Functionalities are grouped into *Features* – objects that represent a prominent aspect that is of interest to the user. A single functionality can implement a complete feature, some aspect of a feature, or fix a bug in a feature. In addition, a functionality can be part of more than one feature.

Figure 3(b) shows a concrete instance of PL-CDM that corresponds to the GlobalCo GPS product line after executing the scenario described in Section 1. Here, the change sets and their grouping into functionalities were produced by hand. In this example, Basic GPS, Trip Computer, Layered Map, Night Mode, POI, Live Traffic Info, 3D Buildings and Shortest Time Routing are features of the GlobalCo GPS product line. They are denoted by rectangular elements. The POI feature is implemented by two separate functionalities, denoted by shaded ellipses. The first one, realized by two change sets (11 and 12), was originally implemented by the team of GPS-Pro and later copied to GPS-EZ. The second one, realized by a single change set (18), was implemented by the team of GPS-EZ when they extended the POI feature.

The *Product Line* object represents the developed product line and *has* a set of *Products* – GPS-Pro and GPS-EZ in our example. Products differ in the functionalities they include. In Figure 3(b), for each functionality in the lower half of the ellipse we list products that include this functionality and strike-through those that do not. E.g., the second functionality of the POI feature (realized by the change set 18) is included in GPS-Pro but not in GPS-EZ.

We say that a product *includes* a feature if it includes *at least one* functionality of that feature. For example, feature POI is included in GPS-Pro because one of its two functionalities is included.

We say that a functionality is *common* if it is included in all products of a product line. A feature is common if each product includes at least one of its functionalities. For the GPS product line in Figure 3(b), POI is a common feature, while the Trip Computer feature is included only in GPS-Pro but not in GPS-EZ.

When a functionality A cannot be realized without a functionality B, we say that A *requires* B. Require dependencies between functionalities are denoted in Figure 3(b) by dashed arrows. For ex-
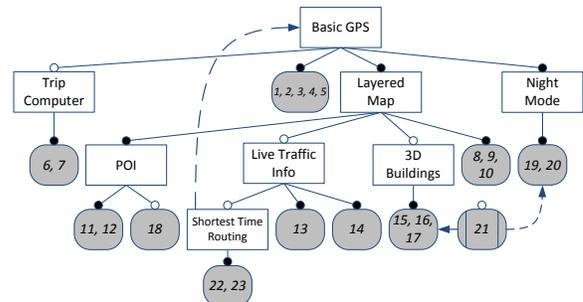

Figure 4: A representation of the GPS product line.

ample, the first functionality of the POI feature requires the single functionality of Layered Map.

Require dependencies between functionalities induce the corresponding dependencies between features: a feature A requires a feature B if at least one of the functionalities of A requires a functionality of B. In Figure 3(b), we determine that feature POI requires Layered Map because the first functionality of POI requires the Layered Map's sole functionality.

Glue (a.k.a. *lifter* in [20]) is a special type of functionality that is used when two or more functionalities need integration. Glue is included in a product whenever all functionalities that it requires are included. For example, the Night Mode and the 3D Buildings features were not designed to work together: the former was implemented as part of product GPS-Pro which does not include the latter. Propagating the Night Mode feature from GPS-Pro to GPS-EZ caused problems presenting three-dimensional buildings when the GPS operates in the night mode. The problem was resolved by creating the glue functionality with change set 21.

3.2 Using the Meta-Model

Explicitly capturing and maintaining the information about the developed products and their corresponding functionalities allows us to construct different *views* on the data, assisting product line managers and developers in their day-to-day tasks.

One example of such a view, describing the complete product line of the GlobalCo GPS products, is shown in Figure 4. In this view, similarly to Figure 3(b), product line features are represented by rectangular elements, whereas their corresponding functionalities are represented by shaded elliptical elements. Similarly to FODA notations for feature modeling [11], we organize the features in a tree structure where parent-child relationships are derived from dependencies between features: a child node *requires* its parent node. When a feature has multiple require dependencies, we pick a random one for the parent-child link and represent the rest as cross-tree constraints (e.g., between the Shortest Time Routing feature and the Basic GPS feature). More sophisticated techniques for heuristically selecting the most appropriate parent element [23] can be applied as well.

We also use FODA notations to represent common and variable features and functionalities: common elements that are part of all products of the product line are marked with a filled circle, whereas those that are part of some but not all products are marked with a hollow one. Even though common features are not necessarily *mandatory* – e.g., it might be possible to produce a product that does not have the Night Mode feature which exists in both GPS-Pro and GPS-EZ, we find the notation useful to describe the *current* state of a product line.

The generated view allows the user to assess the portfolio of existing products (task T1). It also shows that the POI feature is implemented in both GlobalCo's GPS products, but the two implementations are not consistent – one of the products does not contain a functionality that includes change set 18 (task T2).

Using PL-CDM, we can construct another view to show dependencies between features of a particular product, which is especially useful for large models. Due to space limitations, we do not include this view in the paper. However, it can be easily obtained from the representation in Figure 3(b) by including only functionalities and features which are part of the product of interest. This view can tell the developers that if they decide to *remove* the feature Layered Map from GPS-EZ, they should remove the POI feature as well since its implementation depends on Layered Map (task T4). Similarly, when deciding to *add* the Shortest Time Routing feature implemented in GPS-Pro to GPS-EZ, the developers can see that it depends on one of the functionalities of Live Traffic Info, which should be added to GPS-EZ as well (task T3). Furthermore, such a view can support developers in *handling interactions* between features. For example, if the developers decide to copy the 3D Buildings feature from GPS-EZ to GPS-Pro, they will encounter the same issue as faced by the developers of GPS-EZ when integrating the 3D Buildings and the Night Mode features – these feature do not work well together. However, they can detect and reuse the glue that was already created in GPS-EZ.

The information captured in PL-CDM can assist users in a variety of additional tasks, such as identifying the potential consequences of a change (*change impact analysis*) or instantiating new products by selectively "collecting" and assembling features of the existing ones rather than performing a full fork.

4. TOWARDS AN IMPLEMENTATION

In Section 3, we showed that PL-CDM can be used for a variety of product line related tasks performed on forked product variants. While our proposal has not yet been implemented, in this section we discuss ways of collecting the information required to construct PL-CDM and integrating it with contemporary SCM systems.

4.1 Constructing PL-CDM

Figure 2 depicts two activities for building PL-CDM – Feature Aggregation and Feature Dependency Calculation. The input to these activities can come from a variety of sources. Below we describe a few concrete ideas for analyzing the information provided by Change Management and SCM systems.

Feature Aggregation. Most modern SCM systems organize the stored information into *change sets*. To reach a higher level of abstraction, we propose to aggregate change sets into functionalities and features by analyzing meta-data provided by SCM systems. For instance, common branching practices [28, 29] advise creating a separate branch for development of each distinct functional unit. We can then group all change sets committed to that branch into a single functionality and all related functionalities into a feature.

Another source of aggregation information are Change Management (CM) systems, e.g., Bugzilla[1] and IBM Rational Team Concert (RTC)[2]. RTC allows to organize user activities into *work items*

[1] http://www.bugzilla.org/
[2] http://www.ibm.com/software/rational/products/rtc/

which are further grouped into *user stories*. When coupled with an SCM system, a CM system allows associating change sets with work items. Hence, we can group all change sets belonging to the same work item into a functionality and all functionalities belonging to the same user story – into a feature.

Feature location techniques [5], aiming to locate pieces of code that implement a specific program functionality, can also support the aggregation process. Specifically, approaches based on static analysis and information retrieval can be applicable for grouping individual changes into functionalities.

Feature Dependency Calculation. To detect *require* dependencies between functionalities, we can apply static code analysis techniques, e.g., *inter-procedural code slicing* [26], which can help determine whether two pieces of code depend on each other. For example, if code introduced by at least one of the change sets implementing functionality A *modifies* a value of some variable while code of a change set implementing functionality B *reads* this value, we can conclude that B requires A. However, if both functionalities only *read* and *display* the same variable, there is no *require* dependency between them. They both depend on a functionality that *modifies* this variable. In addition, approaches such as [13] can help distinguish semantically meaningful changes from those that introduce *non-essential differences*, i.e., cosmetic in nature and generally not changing the behavior. Such differences can be disregarded when detecting require dependencies.

While static analysis techniques can help identify non-trivial dependencies that are not easily detected by humans, not all dependencies can be detected this way. Additional information can come from the CM meta-data: since work items are mapped onto functionalities, the links between these work items, such as *related to* and *depends on*, can be translated into dependencies between functionalities. Such links are maintained by many CM systems, including RTC.

4.2 Implication for SCM Tools

In our work, we described information necessary for managing forked product variants and assisting the user in a variety of SPLE-specific tasks. This information can be stored in a special-purpose SPLE management layer implemented on top of existing SCM solutions. However, we believe that if the information were available within SCM tools, it could be used for a variety of "generic" (i.e., not only SPLE-specific) tasks. In what follows, we describe several proposed extensions to SCM tools.

Require Dependencies. Existing systems keep track of temporal dependencies between change sets. However, given a change set, they provide no support for determining which dependent change sets are required for the software to function properly. Thus, it is impossible to determine which changes are to be propagated when copying a functionality from one branch to another.

To solve this problem, we propose extending SCM tools with an ability to capture semantic *require* dependencies between change sets. While these dependencies are computed using language-specific code analysis techniques which are hardly appropriate for SCMs, obtaining them from external modules, such as PL-CDM, storing them and presenting them to users when they browse change histories, can support selective propagation of coherent functionalities between branches.

Change Set Propagation. A mechanism called *patching* in RTC and *cherry picking* in Git[3] allows picking and selectively propagating some of the change sets between branches. This mechanism, however, usually results in creating a *new* change set in the target branch, without any traceability to the original one. This leads to divergence of change histories and makes propagation of changes between branches even more complex since users do not have in-

[3] http://git-scm.com/

formation about which functionalities exist in other branches and absent in their own branch.

To address this issue, SCMs could maintain an explicit mapping between change sets as they are propagated between branches. In this paper, we assumed such a mapping by using the original change set numbers in the source and the target branch. For example, in Figure 3(b), the Shortest Time Routing feature copied from GPS-Pro to GPS-EZ is implemented in both products by the same change sets, 22 and 23. This treatment was inspired by a planned extension to RTC that proposes a similar approach for handling history divergence in future RTC releases (work item #170658).

5. CONCLUSION AND FUTURE WORK

In this work, we presented an approach for realizing code reuse as part of SPLE. The approach is based on SCM branching, typically perceived to be the easiest and the fastest reuse mechanism as it provides the ability to start from existing and already tested code, while having the freedom and independence to make any necessary modifications to it.

The proposed approach can be implemented on top of existing SCM systems, allowing the user to reason about the developed product line in terms of features rather than individual code changes made in distinct branches and to detect inconsistencies in implementations of these features. In addition, we proposed to detect and maintain semantic *require* relationships between features, supporting the developers when they copy features from different branches or delete features in their own branch, thus facilitating reuse of features between products.

Instead of maintaining forked product variants, some approaches, e.g., [15, 1], advocate refactoring them into "single-copy" representations, eliminating duplications and explicating variabilities (e.g., the *annotative* or *compositional* SPLE approaches [12]). Explicit guidelines and methodologies for building product lines out of legacy systems have been defined [14, 8], and tool support for helping the user identify similar code elements is available. These tools are based on clone detection mechanisms, e.g., [7, 17], change history analysis [30] and more. Most of the code refactoring approaches, however, are invasive, time-consuming and require significant manual work. Often organizations are not willing or cannot afford the transformation to single-copy representations and thus keep using the forking practices. To support such organizations, we focus on improving the efficiency of practices that are in use rather than attempting to refactor legacy product lines.

However, since our approach explicates product line commonalities and variabilities and traces them to the implementation, it can reduce the transition effort if an organization decides to shift to a different form of code reuse. More research is needed to validate the costs and benefits of supporting existing practices vs. transforming organizations into different approaches. This could be a direction for possible future work.

In addition, our PL-CDM can be further extended to support additional tasks and use-cases, including handling of product releases, as well as detecting and using additional dependencies such as mutual exclusion or feature interactions. Implementing the proposed system and deploying it in a real-life setting is also an obvious direction for future work.

6. REFERENCES

[1] D. Beuche. Transforming Legacy Systems into Software Product Lines. In *Proc. of SPLC'11 Tutorial*, 2011.

[2] M. C. Chu-Carroll and S. Sprenkle. Coven: Brewing Better Collaboration through Software Configuration Management. *SIGSOFT Softw. Eng. Notes*, 25:88–97, Nov. 2000.

[3] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Ser. in Soft. Eng. Addison-Wesley, 2001.

[4] G. Clemm. Activity-Based Software Traceability Management Method and Apparatus, 2010. US Patent 7716649 B2.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *J. of Soft. Maintenance and Evolution*, 23(8), 2011.

[6] N. A. Ernst, S. M. Easterbrook, and J. Mylopoulos. Code Forking in Open-Source Software: a Requirements Perspective. *CoRR*, abs/1004.2889, 2010.

[7] D. Faust and C. Verhoef. Software Product Line Migration and Deployment. *J. of Soft. Practice and Exp.*, 30(10):933–955, 2003.

[8] S. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proc. of SPLC'02*, pages 235–256, 2002.

[9] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-Oriented Version Descriptions in EPOS. *Soft. Eng. J.*, 6(6):378 –386, 1991.

[10] L. Hattori and M. Lanza. Syde: a Tool for Collaborative Software Development. In *Proc. of ICSE'10, Vol. 2*, pages 235–238, 2010.

[11] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI-90TR-21, 1990.

[12] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of GPLE'08 Wksps.*, pages 35–40, 2008.

[13] D. Kawrykow and M. P. Robillard. Non-Essential Changes in Version Histories. In *Proc. of ICSE'11*, pages 351–360, 2011.

[14] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study: Practice Articles. *J. of Software Maintenance and Evolution*, 18(2):109–132, 2006.

[15] C. W. Krueger. Easing the Transition to Software Mass Customization. In *Proc. of 4th Wrksp. on Soft. Product-Family Eng. (PFE)*, pages 282–293. Springer-Verlag, 2002.

[16] C. R. Loff. Graphical Representation of Dependencies Between Changes of Source Code, 2011. US Patent 11/647,905.

[17] T. Mende, R. Koschke, and F. Beckwermert. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):143–169, 2009.

[18] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-Aware Configuration Management. In *Proc. of ASE'09*, 2009.

[19] K. Pohl, F. Guenter Boeckle, and van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, 2005.

[20] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. of ECOOP'97*, pages 419–443, 1997.

[21] A. Sarma, Z. Noroozi, and A. V. D. Hoek. Palantir: Raising Awareness among Configuration Management Workspaces. In *Proc. of ICSE'03*, 2003.

[22] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proc. of SPLC'10*, pages 77–91, 2010.

[23] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proc. of ICSE'11*, 2011.

[24] M. Staples and D. Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proc. of APSEC'04*, pages 176–183, 2004.

[25] C. Thao, E. Munson, and T. Nguyen. Software Configuration Management for Product Derivation in Software Product Families. In *Proc. ECBS'08*, pages 265–274, 2008.

[26] F. Tip. A Survey of Program Slicing Techniques. *J. Prog. Lang.*, 3(3), 1995.

[27] J. van Gurp and C. Prehofer. Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families. In *Proc. of SPLC'06 Wrksp. on Variab. Mgment*, pages 48–58, 2006.

[28] C. Walrad and D. Strom. The Importance of Branching Models in SCM. *IEEE Computer*, 35(9):31–38, 2002.

[29] L. Wingerd and C. Seiwald. High-level Best Practices in Software Configuration Management. In *Proc. of SCM'98*, volume 1439 of *LNCS*, pages 57–66, 1998.

[30] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE: Factor Analysis Based Approach for Detecting Product Line Variability from Change History. In *Proc. of MSR'08*, pages 11–18, 2008.

[31] A. Zeller and G. Snelting. Unified Versioning Through Feature Logic. *ACM Trans. Softw. Eng. Methodol.*, 6:398–441, 1997.