# Aligning Coevolving Artifacts Between Software Product Lines and Products


Sandro Schulze
TU Hamburg-Harburg
sandro.schulze@tuhh.de

Michael Schulze, Uwe Ryssel
pure-systems GmbH
{michael.schulze, uwe.ryssel}@pure-systems.com

Christoph Seidl
TU Braunschweig
c.seidl@tu-braunschweig.de



## ABSTRACT

Software product lines (SPLs) play a pivotal role for developing a vast amount of related programs efficiently and with high quality. To this end, the SPL engineering process is separated into two levels: domain engineering (DE), which captures variability and development artifacts of the entire SPL, and application engineering (AE), which encompasses a variant-specific subset of the aforementioned artifacts. In the industrial practice of evolving an SPL, it is common that evolution is performed on both levels, which may affect the same artifacts (e.g., code, models) in different ways due to changes on the product line (DE) and the variant level (AE). As a result, conflicts may arise that have to be solved properly to guarantee correctness and validity of the affected artifacts. In this paper, we propose a methodology for resolving such conflicts to ensure correctness and consistency among artifacts while minimizing manual effort. Our method is comprehensive in two ways: First, we consider all kinds of artifacts (code and non-code) that may be subject to evolutionary changes in both DE and AE. Second, we also take into account that changing one particular artifact (e.g., a requirement) may require further changes to other artifacts of the same level. This way, our method reflects common industrial practices in SPL development and, thus, provides benefits for efficiently evolving real-world SPLs.



## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*

## Keywords

SPL engineering, software evolution



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.



*VaMoS '16, January 27 - 29, 2016, Salvador, Brazil*



© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.



ISBN 978-1-4503-4019-9/16/01...$15.00

DOI: http://dx.doi.org/10.1145/2701319.2701329


## 1. INTRODUCTION

Software evolution is a common phenomenon for *all* software systems when they are subject to ongoing changes over time due to new or altered requirements [6]. As a result, software development is a dynamic, continuous process rather than a singular task. Software systems have to undergo maintenance, testing, refactoring and other quality-related tasks in order to preserve a clear and evolvable structure. *Software product lines (SPLs)* recently gained importance due to their intrinsic ability to support mass customization of software. In particular, SPL engineering allows managing, developing and maintaining an entire family of similar, yet well-distinguished, program variants of a certain domain [2, 10]. Consequently, SPLs constitute a long-term investment for companies so that they also have to undergo a continuous and laborious process of software evolution [6, 14, 7].

The SPL engineering process may be divided into two levels: *Domain engineering (DE)* is responsible for defining the common and variable parts of the SPL, usually specified by a *variability model* [3]. In contrast, *application engineering (AE)* is responsible for deriving a specific variant according to the needs of a particular customer. Due to this distinction, evolution of SPLs takes place on two different levels [7]: Evolution on the DE level affects multiple variants that can be derived according to the variability model. For instance, adding improvements, such as more efficient algorithms, that may be beneficial for a large amount of variants are usually implemented on this level. In contrast, evolution on the AE level affects only artifacts of one particular variant. For example, smaller adaptations related to the underlying, customer-specific platform or specific requirements posed by individual customers are usually easier and faster to accomplish on AE level than in DE. Especially in practice, it is common to evolve particular variants *after* they have been derived from the common platform [2].

Given these two levels of SPL engineering, the same artifacts may evolve differently at the same time as part of the product line (DE) or individual variant (AE). As a consequence, this process of individual evolution breaks with the concept of *coevolution*–a fundamental principle that ensures the consistent evolution of corresponding artifacts.

Hence, the affected artifacts may contain conflicting information regarding the DE and AE levels. This is problematic for further development: Either the variant cannot be regenerated from domain artifacts, which neglects the changes of DE, or the variant is regenerated, which discards the changes

of AE by overriding them.[1] The prior is problematic as variants may not benefit from the changes to the entire SPL. The latter is problematic as the changes made to a particular variant in AE are usually important adaptations and, thus, should not be overwritten. Instead, a more fine-grained approach that analyzes and merges the changes of both levels would be beneficial. Currently, there is no such automated solution to resolve this conflict by preserving changes of both DE and AE. Instead, either the evolutionary changes of a particular level are neglected or developers have to resolve conflicts manually, which is infeasible for real-world applications.

In this paper, we go beyond current research, which considers evolution of SPLs only on DE or AE level separately, mainly for source code or variability model artifacts (e.g., [8, 9]). In contrast, we provide a methodology that explicitly addresses this separated coevolution on both levels, considering all artifacts of software development such as requirements, source code, models and so on.

**Problem:** Coevolution of development artifacts in both, DE and AE, is common in SPLs. As a result, particular artifacts may evolve separately, leading to conflicts when attempting to regenerate the corresponding variant from domain artifacts. Current methods for evolving SPLs do not tackle these conflicts.

**Contribution:** We propose a comprehensive methodology that addresses both, detecting and resolving *syntactical* conflicts, arising during (co)evolution of SPLs. In particular, we provide a conceptual framework that defines possible conflicts in general (i.e., independent of concrete artifacts), also considering different levels of granularity. Moreover, we integrate our approach with the common SPL engineering process, thus increasing acceptance in industry. We argue that our methodology lays the foundation for a more efficient and automated evolution of SPLs that can be applied to industrial practice.

The remainder of the paper is structured as follows. In Section 2, we present a running example used to illustrate the challenges and solutions of the paper. In Section 3, we contribute a classification scheme for conflicting changes that may or may not be resolved automatically. We further present a mechanism to detect the respective types of conflicts as well as to resolve them where possible. In addition, we propose a conceptual integration of our solutions in existing SPL engineering processes. In Section 4, we present the realization of our concepts within the tool pure::variants as well as initial experiences from industry. In Section 5, we discuss related approaches and in Section 6, we close with a conclusion.

## 2. RUNNING EXAMPLE

To illustrate the process and resulting problems of coevolution between DE and AE, we introduce a running example that is used throughout this paper to illustrate our concepts. Imagine an automotive front lighting system SPL for passenger cars. Requirements (on different specification levels), models (e.g., UML, Matlab/Simulink, AUTOSAR), source code, tests, and variability models forming the system's product line from which variants are derived. The abstract view on the derivation and also on the evolution process is shown in Figure 1. The depicted *Product Line Assets* boxes act as placeholders for all the different artifacts and each *Variant A* box represents all artifacts belonging to variant A.

[1]Note that generation in this sense not only focus on generating code or the documentation. Rather, this also includes other configuration-specific aspects such as calibration of certain parameters or define binding times.

According to Figure 1, the creation of a customer specific variant A starts at $t_0$ with the derivation step, which is symbolized in the figure by ①. This step basically consists of multiple actions such as:

- *feature selection* according to the customer-specific requirements. As a result, we obtain a concrete feature configuration, which is also part of the created variant.
- *transformation* of artifacts corresponding to the selected features. Among others, composing respective requirements or code artifacts but also calibration of parameters are examples for this action.
- Finally, a concrete variant of the system is generated based on the previously transformed artifacts.

As result of this step, we obtain a working copy for the derived variant. This variant constitutes the base for further development as, usually, the SPL is not able to deliver the entire functionality customers demand due to individual requirements. Hence, changes of particular artifacts, such as add, remove, and modify, take place on the derived variant (AE). This leads to a customer-adapted and, usually, functionality enriched variant (represented as *Variant A'* in Figure 1).

Besides modifications on variants' working copies, changes also take place on the entire product line (i.e., DE level), e.g., through maintenance activities such as bug-fixing or functionality extension in order to satisfy emerging market-needs. The changes on both layers are realized simultaneously and in an unsynchronized manner (marked with ② in the figure). In general, this is not a problem and often even desired in industry as it allows variants of different customers to develop with their own speed.

However, a problem arises if a derived variant requires further functionality or bug-fixes from the SPL (DE). In this case, the aforementioned derivation process is performed again at $t_1$ (step ③), which leads to the generation of a new working copy for the variant. All changes performed for the variant (AE) on artifacts stemming from the SPL, independent of the kind of change, are overwritten by the artifact versions of DE level. The potential loss of essential changes performed on AE level (visualized by scissors in the Figure 1) is a major concern for real-world SPLs due to the resulting increased cost of recreating the changes and the possibly reduced quality of the solution.

## 3. METHODOLOGY

Given the evolutionary conflicts described in the previous section, a methodology is needed that a) detects such conflicts and b) supports developers in solving these conflicts–where possible automatically. In this section, we provide a conceptual framework, which addresses both aspects.

First, we systematically define conflicts between DE and AE that may arise during SPL evolution independent of the concrete type of development artifact. Afterwards, we propose a corresponding resolution process, which defines particular actions to be taken for detecting and resolving the introduced conflicts.

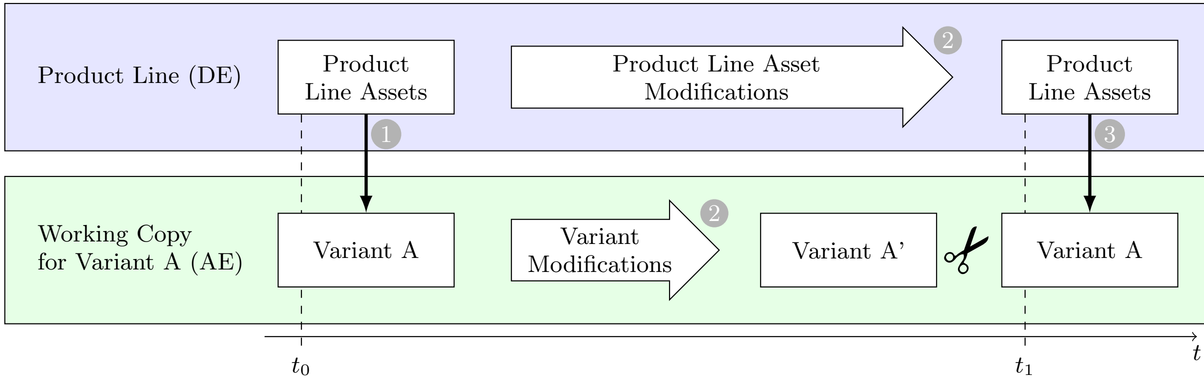


**Figure 1: Mixing variant and product line evolution–challenge.**

## 3.1 Classifying Co-Changes in SPLs

Change is an inherent process to software in order to conform with new or altered requirements, environments, etc. [6], which also affects SPLs. To perform evolution, we distinguish three basic operations that can be part of an evolutionary task, independent of the affected artifacts:

- *add:* An artifact (e.g., a requirement, code, model etc.) may be added, e.g., to extend functionality.
- *remove:* An artifact may be removed, e.g., because it is no longer needed.
- *modify:* An artifact may be adapted according to changing circumstances, e.g., due to legal issues.[2]

In their general nature, the aforementioned operations also apply for SPLs. However, the actual change process takes place on both, DE and AE level. Hence, a conflict may arise when a particular artifact is changed on both levels concurrently but in different way, which we refer to as *conflicting co-changes*. Next, we explain how and when such changes may occur.

### *Conflicting Co-Changes.*

Basically, we define a conflicting co-change as a binary operation that occurs between two changes that occur independently on DE and AE level, respectively, but on the same artifact type (e.g., code, model) and on the same level of granularity (e.g., file, block, line). As these conflicts harm SPL integrity, it is essential to resolve them. As a first step, it is imperative to raise awareness *when* such conflicts may arise and to what extent they can be resolved automatically or not. As an essential step, we first have to identify change operations that are likely to produce conflicting changes and how they can be resolved. Such a priori knowledge makes it more efficient to detect conflicts as it supports developers in identifying those changes that need specific treatment. To this end, we contribute a *conflict matrix*, shown in Figure 2 that puts domain and application engineering in relation with respect to the aforementioned change operations .

As a result, the matrix illustrates potential conflicts for combinations of particular change operations when performed on DE and AE level concurrently. Hence, each change operation of one level is compared with all change operations of the other level regarding the possibility of conflicts. Moreover, the color of the matrix cells indicates whether such a conflict is automatically resolvable or not.

[2]While this operation can be considered as a combination of the remove and add operations, its semantics are important for determining conflicts. Hence, we treat this operation separately.

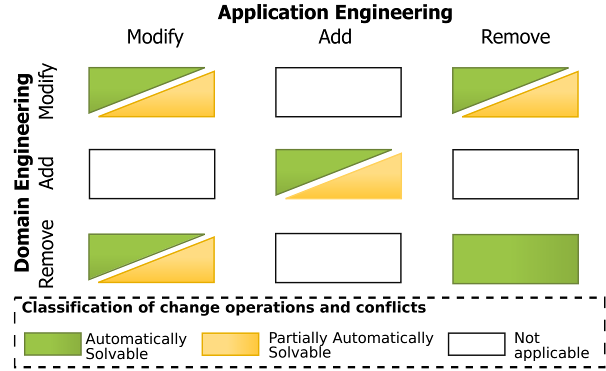


**Figure 2: Conflict matrix that indicates conflicting change operations between AE and DE as well as whether they can be resolved automatically.**

Given the conflict matrix, we identified three categories of (possibly) conflicting change operations. First, in four cases actually no conflicts can occur that have to be resolved (black, empty boxes in Figure 2). The reason is that the respective change operations of DE and AE level do not affect each other (i.e., they are orthogonal). For instance, if on DE and AE an artifact of a particular granularity is added (e.g., a line of code) this will not cause any conflict by our definition. As an example, consider Figure 3 as an excerpt from MATLAB/Simulink models of our automotive example, introduced in Section 2. Omitting the remove operation of *Change A* and modify operation of *Change B*, the add operation of both changes are not in conflict because they take place on different locations despite modifying the same file. Hence, such changes could be merged with no or only low effort by automated merging techniques.
Second, for another four cases, the respective change operations will result in a conflict which needs to be resolved. For instance, if the same location on the same artifact is modified, this will inevitably lead to conflicts that cannot be resolved by a simple textual merge, thus, more sophis-

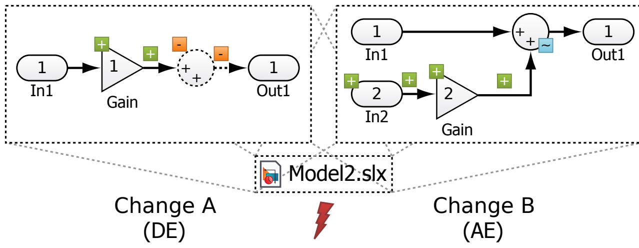


**Figure 3: Change operations may be in conflict.**

ticated techniques are needed. In our example in Figure 3, the SUM block is changed in different ways, i.e., removed on DE level (*Change A)* while the same block is changed on AE level (*Change B*) by adding a port to connect new blocks to it. In such cases, more advanced techniques are necessary to address these conflicts, particularly to obtain the required information to guide developers in making a decision.
Finally, we argue that removing artifacts on DE and AE level does not result in any conflict. For instance, removing different or even the same blocks from the same model on DE and AE level, respectively, can be done without interference.

While our matrix mainly illustrates the interrelation of change operations at the same grain, the *granularity* of a change itself plays a pivotal role detecting possible conflicts. We illustrate this aspect with an example, shown in Figure 4. On the left side of the figure, we show an excerpt of the directory structure, containing two *.slx* files (representing possible MATLAB/Simulink models). Moreover, the "∼" on the file `Model2.slx` indicates a *modify* operation to this file on the *file system level.* However, if we consider the right side of Figure 4, the changes made on *artifact level* actually correspond to *add* operations. Hence, it is important to take the granularity of changes into account for classifying changes. In our approach, we always consider the most fine-grained changes for classifying them according to our matrix. Hence, in our example, we would classify the changes as add operations. Interestingly, this differentiation also enables us to reuse existing techniques from software merging, especially for coarse-grained level changes. For instance, if *add* or *remove* operations take place on file system level, we can rely on established techniques also used in current version control systems (cf. Section 4 for more details).

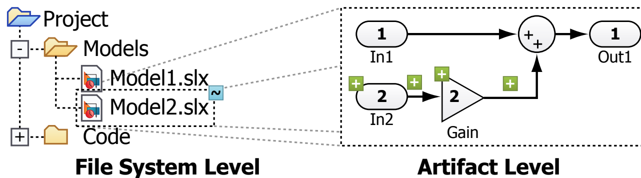


**Figure 4: Change operations may appear differently on file system and artifact level.**

*Cascading Co-Changes.*

While conflicting changes are already challenging for the same artifact type, the problem may be even amplified by the following situation: a change operation *within* one of the respective levels may trigger changes to further, related artifacts of the same level. We refer to such changes as *cascading co-changes.* As an example, consider a change operation (e.g., add) to the requirements or the architecture in DE (i.e., SPL assets in Figure 1). As a result of such a change, it is likely that corresponding code fragments have to be changed as well. Consequently, if the initial change operation results in a conflict (regarding AE level), the cascading co-changes may do so as well. Hence, the set of change conflicts and, thus, the effort for resolving them, may increase rapidly even for few (initial) changes.

Next, we will propose our concept for resolving such conflicts in order to merge concurrent changes between AE and DE. Note that we consider only merging changes from DE level into AE level during product derivation while the inverse direction is subject to future work.

## 3.2 Detection of Conflicting Co-Changes

Based on the introduced conflict matrix, we gain understanding of different conflicts with respect to their resolvability. However, to classify changes according to our matrix, we first need to detect them. As a first, naive approach, we could simple use the evolved versions on DE and AE level and compare them. Regarding Figure 1, we would compare Variant A' as the evolved working copy (i.e., AE) and Variant A at time $t_1$ that constitutes the newly derived working copy from evolved product line assets (i.e., DE). As a result of such a *2-way comparison*, we obtain all differences between both variants. However, the results do not provide us with information that can be used to classify these differences in terms of change operations. The reason is that with a 2-way comparison, it is neither possible to determine which of the variants has been changed nor what change operation has been performed for a particular difference. Hence, detecting conflicts between these two variants is impossible.

The problem of the aforementioned approach is that it misses a *common base* to compare both of the variants with. Regarding our example in Figure 1, the original working copy (i.e., Variant A at time $t_0$) constitutes this common base from which both variants originate. Given this common base, we can now employ a more advanced approach to obtain the changes between DE and AE by a *3-way comparison.* With this approach, we compare the evolved variants of DE and AE level not only with each other, but also with their origin, i.e., the common base at time $t_0$. As a result, we can precisely determine which change operation has been applied to the respective variant. Consequently, we can classify the changes according to our matrix and, thus, identify possible conflicts.

## 3.3 Integration with SPLE Process

As described above, we have to apply a 3-way comparison on the artifacts of different versions of SPL variants. Software configuration management (SCM) systems, such as git[3] or Subversion[4], usually offer a 3-way comparison for merging concurrent modifications from a branch to a working copy. In the SCM domain, these three compared versions are referred to as *base*, *theirs* and *mine* versions. The *base* version is the common ancestor, which is modified by the other two versions. The *theirs* version is the modified version in the branch. The *mine* version is the modified local copy, where the changes of the branch should be integrated.

[3]http://git-scm.com
[4]https://subversion.apache.org

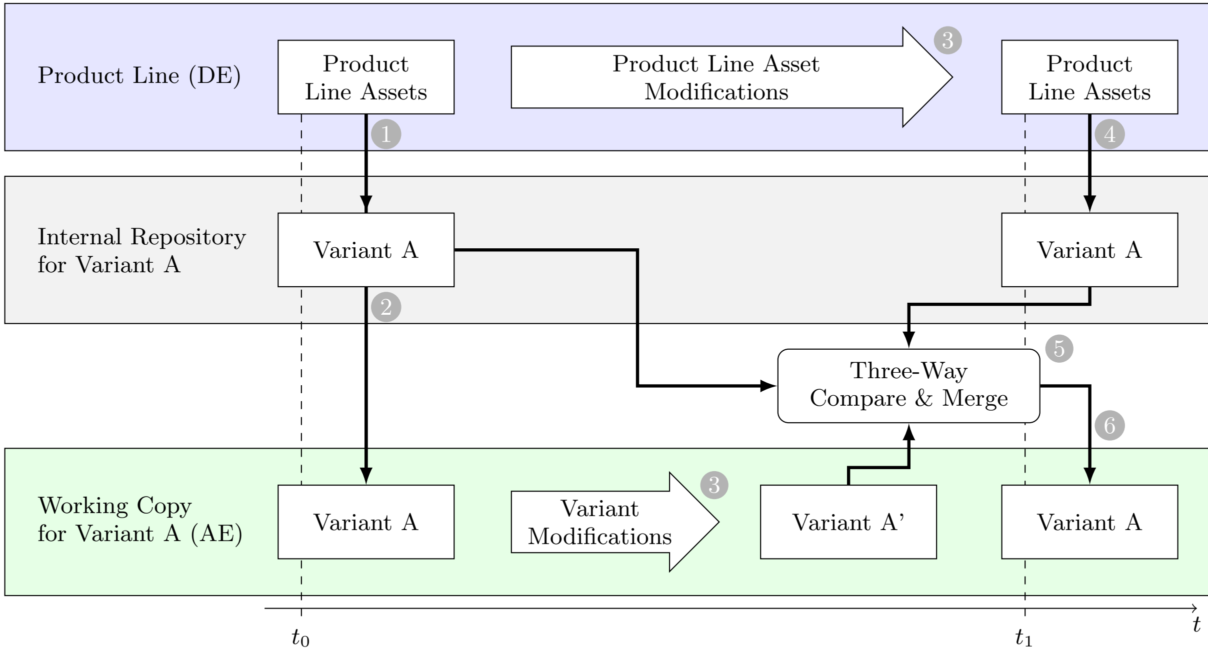


**Figure 5: Mixing variant and product line evolution–implementation.**

However, in our SPL engineering scenario, not all of the three needed versions are available explicitly. Basically, only the *mine* version is available as the application engineer's current modified version $A'$. Moreover, the *theirs* version can be generated from the SPL artifacts at their current state. However, retrieving the *base* version is more sophisticated because information is needed that is (usually) no longer available to application engineers. Generally, two approaches are conceivable to solve this problem as follows.

In the first approach, the *base* version is regenerated from the SPL. However, this requires a snapshot of the SPL including the generators employed for the point in time the previous *base* version was generated (e.g., time $t_0$ in Figure 1. This also requires that the generators are deterministic so that the generated products are always identical for the same configuration. Provided that the SPL is published in fixed release versions, these snapshots should be easily retrievable even if application engineers have no access to interim versions. However, if there are no such release versions, a snapshot of the entire SPL has to be generated each time a generation process is triggered on a changed SPL.

In the second approach, each generated variant is saved in a, possibly local, repository to keep it for later use. This approach is shown in Figure 5. Between the two known levels of the SPL and the working copy of a specific variant, a new level for the repository is introduced, which is transparent for application engineers. When application engineers derive a specific variant $A$ for the first time (specified as $t_0$ in the figure), it may be saved automatically in a repository, which is responsible for that variant (step 1). The working copy retrieved by application engineers can be cloned from that version (step 2). Over time, the SPL and the working copy can be changed independently from each other (step 3). At the later point in time $t_1$, application engineers may want to update their working copy to the current SPL version. During that automatic update process, a current version of variant $A$ can be derived from the SPL and saved directly in the repository (step 4). This version is not shown to application engineers. Instead, a 3-way comparison and merge is performed among the two versions in the repository (the old version as *base*, the current version as *theirs*) and the working copy as *mine* version (step 5). The merge can be done mostly without user interaction. Only for conflicts that are unresolvable with regard to 2, application engineers have to decide, which changes should be applied. The result is an updated working copy with merged changes of the DE and AE level (step 6). This update process can be repeated each time the SPL is changed. In that case, the input of the three-way comparison is the saved repository version of the last update as *base* version, the current repository version as *theirs* version and the working copy as *mine* version.

In the next section, we provide details about the implementation of our structured 3-way comparison and how it aligns with the SPL engineering process. In particular, we explain how we derive *base*, *theirs*, and *mine* versions from DE and AE.

## 3.4 Applicability and Limitations

Basically, our proposed classification scheme is general enough to be applicable with different scenarios and different artifacts in SPL development. The reason is that our definition of both, change operations as well as change conflicts, is *artifact-independent* and that we address the integration in the common SPL development process. However, due to its general nature, our method requires some manual effort to be adapted for concrete product lines. Most importantly, the concrete artifacts must be defined that are subject to change operations and an instantiation of their *granularity levels* must be provided. The latter is of specific importance, because the granularity plays a pivotal role to decide whether a conflict exists or not. Moreover, granularity levels are different for particular artifacts. For instance, for source

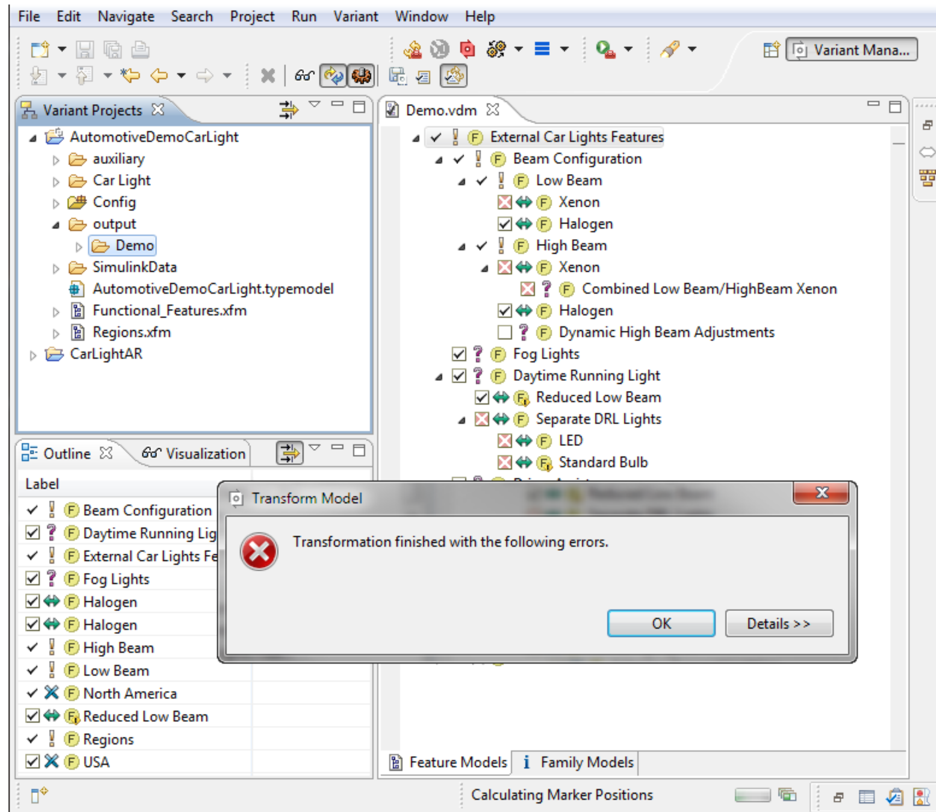


**Figure 6: Updating a variant of the car lighting example in pure::variants.**

code it may be sufficient to distinguish between statement, block, and file level. In contrast, considering artifacts of hierarchical structure, such as requirement specifications, different levels of granularity such as line, section, or subsection may be required to detect conflicts with a suitable accuracy. Finally, developers have to specify how the conflict detection and resolution is integrated in the (most likely already existing) development process, for instance, which tools to be used for conflict detection. However, the aforementioned instantiation has to be done only once (when setting up or integrating with an existing SPLE process) and subsequently can be used for the entire evolution process.

Finally, it is worth to mention that, with our proposed classification, we mainly focus on *syntactical* changes. As a result, our classification does not ensure *semantical* correctness. For instance, in Figure 3, the add operations are syntactically not in conflict but may result in a semantically different model (e.g., Change B may lead to a different result of the SUM block). However, we argue that syntactical correctness is the stepping stone for consistent co-evolution in SPLs and, thus, for ensuring integrity of both, DE and AE level.

## 4. IMPLEMENTATION & APPLICATION

In this section we show, how our methodology can be applied in practice. In our prototypical implementation, we have integrated the described process into pure::variants[5], an industrial variant management tool, which supports the development of product lines. This tool can manage different types of realization artifacts, either by generic modeling in the tool or by integration into external tools using specific connectors. The derivation process of variants is handled by an extensible set of transformations, which are specific to the artifact type or external tool. These transformations are the connection point for our implementation.

The prototypical implementation currently supports source code and requirement artifacts. The internal local repository is realized by a git repository. For source code, git is also used for the 3-way comparison. The actions of saving each derived variant in the repository and of executing the 3-way comparing among the repository and working copy versions were integrated transparently in the transformation process for source code variant derivation. So when the application engineer wants to update his working copy, he just starts a new derivation of the current variant. If there are no conflicts, which have to be resolved manually, the application engineer will get the merged result without further user interaction. If there are such conflicts, the transformation outputs an error (see Figure 6). For each conflict, a conflict resolution view is shown to the application engineer. Figure 7 shows an example for a source code artifact with such a conflict: In file `output_foglight.c` the same line is modified in the current SPL variant and in the working copy. The application engineer has to decide, which version he prefers to be in the merged result. At the end, the application engineering gets a merged version semi-automatically, without even knowing that a local repository layer exists.

[5]http://www.pure-systems.com

**Figure 7: Merge view for a source code artifact with a conflict.**

The prototypical implementation was presented to different customers, which gave us a positive response. For many customers the support of the described use case of updating co-evolved working copies is of high importance. Hence, our methodology is of high relevance in the industrial domain.

# 5. RELATED WORK

In this section we acknowledge related work and point out how it is different compared to our technique.

## 5.1 Product-Line Evolution

Different aspects of SPL evolution has been subject to research in previous work.

Passos et al. propose evolution patterns for the coevolution of variability model and source code artifacts by means of the Linux kernel [9]. However, this approach is limited to domain engineering only and considers only two different artifact types. Moreover, it supports analyzing the evolution in the part while not addressing how to evolve SPLs in future.

Seidl et al. consider models as development artifacts and their mapping to features in the feature model [13]. To this end, they propose concrete operations to be applied to keep these mappings consistent on DE level. However, these operations are tailored to the concrete artifact types and thus, cannot be applied to other artifact types. Furthermore, the authors assume a valid state of the SPL before and after the operations are applied but cannot deal with conflicts. Moreover, they do not consider coevolution between DE and AE level.

Botterweck et al. propose a tool-supported solution, *EvoFM*, for the proactive evolution of SPLs, in particular for model-based product-line engineering [1]. To this end, they provide concrete evolution operators and mappings from model artifacts to features. Similarly to the aforementioned approach, this approach is restricted to dedicated artifact types and mainly considers evolution on DE level. Nevertheless, they also integrate their approach in an existing SPL engineering process as we propose as well in our methodology.

Vierhauser et al. focus on consistency checking of variability models in SPLs [15]. In particular, they propose different categories of possible inconsistencies. To this end, they consider problem and solution space of SPLs, but focus only on domain engineering. Moreover, they do not propose how to resolve these inconsistencies as we do for conflicting changes.

Jirapanthong et al. present an approach to automatically identify semantical traceability links (e.g., refines, implements) between SPL artifacts [5]. However, their approach is currently limited to a predefined set of artifacts, which conflicts with our goal of generic solution space assets that supports arbitrary artifact types.

To summarize, our proposed method differs substantially from related work in two ways. First, our propose methodology is developed to support a variety of artifact types and thus, support the whole software development lifecycle, that is, from requirements to test and implementation. Second, we take both levels, domain and application engineering, into account which is essential to support common industrial practice in SPL evolution (i.e. the fact that changes occur on both levels). Moreover, we integrate our method into the product-line engineering process comply with industrial practice and thus, to support applicability regarding common processes.

## 5.2 Software Merging

Besides coevolution, our proposed method method relates to software merging. While numerous approaches exist, addressing this topics, we restrict our considerations of related work to this that apply merging (or even differencing) in the context of SPLs.

First, Rubin et al. proposed different operators, including merging for consolidating different variants with a common base to SPLs [11] and applied them in three different case studies [12]. Compared to our method, there are two main differences. First, Rubin aims at consolidation and thus, assumes that variants have been evolved independently over a considerable time. Hence, the operators are designed to detect commonalities and variabilities, using this information to finally merge the variants. Second, she proposes a dedicated framework for this process. In contrast, we consider already existing SPLs and a corresponding SPLE process. Moreover, we integrate our method in such an existing process.

Next, Duszynski et al. proposes a differencing algorithm to analyze multiple related variants for commonalities and differences [4]. However, he uses a simple, text-based algorithm and only applies it to source code. In contrast, we

focus on different artifact types and try to exploit artifact-specific information as well.

# 6. CONCLUSION AND FUTURE WORK

In this paper, we identified the need to merge conflicting changes of SPL evolution that result from simultaneous changes of the same artifacts on DE and AE level. We presented a classification of change operations, how these operations may lead to conflicts between domain and application engineering, and which conflicts can be resolved automatically or need manual intervention. While this conflict detection needs to instantiated properly for a concrete application scenario, it comprises all kinds of configuration-related activities, that is, not only selecting features and generating code, but also dealing with different binding times and calibration of respective parameters. This way, our proposed method can be integrated into common SPLE development processes. We further provided a general process for identifying conflicting changes with respect to their category. Furthermore, we contributed a procedure based on 3-way merging that allows automatic resolving of conflicts where possible. We integrated our concepts in the industrial tool pure::variants and demonstrated their applicability in an exemplary application within an industrial scenario.

In future work, we plan to implement our methodology also for other kinds of structured artifacts, such as UML models. For a generic implementation, we have distinguished two challenges: One challenge is to get a high-quality 3-way comparison for any kind of structured artifacts, which can be hierarchical, graph-oriented, ordered or unordered. The second challenge is to get support for artifact-specific data, which should be considered in comparison and merge. An example are *changed*-flags, which can be ignored in comparison, but have to be set properly in the merged working copy to only reflect real changes. Furthermore, data that is linked to the working copy has to be protected against deletion during the update process. Finally, we will evaluate our concepts and their implementation in an industrial setting.

# 7. ACKNOWLEDGMENTS

This research has been supported by the Federal Ministry of Education and Research in project SPES XT (funding id: 01IS12005). The responsibility for the contents rests with the authors.

# References

[1] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. "EvoFM: feature-driven planning of product-line evolution". In: *Proceedings of the Workshop on Product Line Approaches in Software Engineering (PLEASE)*. ACM, 2010, pp. 24–31.

[2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing, 2001.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[4] S. Duszynski, J. Knodel, and M. Becker. "Analyzing the Source Code of Multiple Software Variants for Reuse Potential". In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 303–307.

[5] W. Jirapanthong and A. Zisman. "XTraQue: Traceability for Product Line Systems". In: *Software & Systems Modeling (SOSYM)* (2009).

[6] M. M. Lehman. "Programs, Life Cycles, and Laws of Software Evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076.

[7] T. Mens, A. Serebrenik, and A. Cleve. *Evolving Software Systems*. Springer, 2014.

[8] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. "Feature-oriented Software Evolution". In: *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 2013, 17:1–17:8.

[9] L. Passos, L. Teixeira, D. Nicolas, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo. "Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel". In: *Empirical Software Engineering* (2015). To appear.

[10] K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[11] J. Rubin and M. Chechik. "A Framework for Managing Cloned Product Variants". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1233–1236.

[12] J. Rubin, K. Czarnecki, and M. Chechik. "Managing Cloned Variants: A Framework and Experience". In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2013, pp. 101–110.

[13] C. Seidl, F. Heidenreich, and U. Aßmann. "Co-Evolution of Models and Feature Mapping in Software Product Lines". In: *Proceedings of the International Software Product Line Conference (SPLC)*. 2012.

[14] M. Svahnberg and J. Bosch. "Evolution in Software Product Lines". In: *Journal of Software Maintenance and Evolution* 11.6 (1999), pp. 391–422.

[15] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. "Flexible and Scalable Consistency Checking on Product Line Variability Models". In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 2010.