Semantics-Based Version Control for Feature Model Evolution Plans

Eirik Halvard Sæther[1], Ingrid Chieh Yu[2], and Crystal Chang Din[3]

[1] Norsk Rikskringkasting (NRK), Norway
eirik.halvard.95@gmail.com
[2] University of Oslo, Norway
ingridcy@ifi.uio.no
[3] University of Bergen, Norway
crystal.din@uib.no

Abstract. A software product line (SPL) models closely related software systems by capitalizing on the high similarity of the products by organizing them into common and variable parts. To ensure successful long-term development, it is beneficial to not just capture the current software product line, but also the planned evolution of the SPL as well. Evolution planning of an SPL is often a dynamic, changing process due to changes in product requirements. In addition, planning is typically a collaborative effort with multiple engineers working separately and independently of each other. To improve development, their individual contributions would need to be unified. This can be a complex task, especially without proper synchronization tools. In this paper, we provide a semantics-based merge algorithm for evolution plans. Given two versions of an evolution plan and the common evolution plan they are derived from, the merge algorithm attempts to merge all the different changes from both versions. The merge algorithm will be an essential component in a version control system, allowing several contributors to unify their versions into a sound evolution plan.

Keywords: Software Product Lines, Software Evolution, Evolution Planning, Version Control

1 Introduction

A Software Product Line (SPL) is a collection of closely related software products. The different software products leverage the high similarity by explicitly encoding the common and variable parts [10,22]. As a software development paradigm, SPLs allow the engineers to model the software's commonality and variability to facilitate large-scale reuse. The most common variability model, feature models (FMs) capture the different aspects and visible characteristics of a system in terms of features [23]. The feature model organizes the features in a tree structure to capture the variability of the SPL. Each feature corresponds

to a certain increment in program functionality. The particular variant of the software product line is defined by a unique combination of features [6].

Due to changes in product requirements, planning for a long-term SPL evolution is often crucial [5,7,9,21]. Product-based evolving SPL was discussed in [11], where the evolution first happened at the product level, and to be later merged back into the SPL platform where the core assets reside. Hoff et. al. [14] performed semantics-based soundness checking on a given evolution plan defined as a sequence of FMs. The plan captures not only the current FM but all intended future FMs and checks if the plan will be applicable. However, *planning* the evolution of an SPL before the product is built often involves multiple engineers and the plan can be changed over time according to the updates of the product requirements. Hence, enabling several engineers to work in parallel on the *changes* of the same evolution plan can be beneficial in handling the dynamic nature of evolution planning. This means that their individual contributions must be unified into a sound evolution plan. Generic version control systems, such as Git, have been extensively used to synchronize the efforts of multiple programmers. It is a text-based approach and the semantics of the language is not considered during merging. Naively merging multiple versions of an evolution plan easily yields inconsistencies and conflicts. The semantic correctness of merging evolution plans is heavily dependent on time (e.g., which time window a future feature must become (un)-available) and space (the structures of the FM in the future). In this work, we investigate merging strategies that respect the structure and semantics of evolution plans and can detect various merging errors, addressing the limitations of current tools.

The paper is organized as follows: Section 1.1 defines feature model evolution plans and their soundness, and Section 1.2 provides a motivating example. Section 2 describes the algorithm of our semantics-based merger for feature model evolution plans. Finally, we discuss related work and conclude the paper in Section 3.

1.1 Feature Model Evolution Plans

Software product line engineering is a discipline for efficiently developing such families of software systems. Instead of maintaining potentially hundreds of different software variants, these engineering methods have ways of capitalizing on the similarities and differences between each variant. Variants of a software product line can be defined in terms of a feature model (FM). These models play a central role in planning and are also used as a communications tool. A feature model is a tree structure of features and groups. Features represent a concrete aspect in an SPL. Features can be mandatory or optional and will contain zero or more groups. Each group has a set of features and the type of the group dictates which features in the group can be selected. In an *and* group, all the mandatory features have to be chosen, while *or* groups have to select at least one feature and *alternative* groups have to select exactly one feature. Therefore, groups with types *or* or *alternative* must not contain mandatory features. We use ● to indicate mandatory features, ○ to indicate optional features, ∧ for *and*

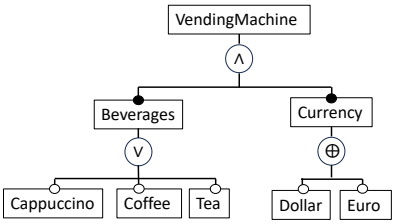

Fig. 1. Vending machine feature model

group, $\vee$ for *or* group, and $\oplus$ for *alternative* group. Note that this explicit representation of group types helps the users compare the group type differences and see the result of merging. An example of a vending machine feature model can be found in Fig. 1. All the vending machines configured based on this feature model should provide at least one type of the beverages (Cappuccino, Coffee or Tea) for the customers to choose from. The features Dollar and Euro are in an *alternative* group, indicated by the $\oplus$ symbol. This means either the vending machine is designed to accept Dollar or is designed to accept Euro (but not both).

Wellformedness Requirements The wellformedness requirements of a feature model include not only the structural restrictions mentioned above but also the following list of constraints. These wellformedness requirements define what is considered to be a sound feature model: (1) feature model has exactly one root feature, (2) the root feature must be mandatory, (3) each feature has exactly one unique name, variation type and (potentially empty) collection of subgroups, (4) features are organized in groups that have exactly one variation type, (5) each feature, except for the root feature, must be part of exactly one group, and (6) each group must have exactly one parent feature.

Definition 1 (A Sound Feature Model Evolution Plan). *A paradox is a violation of the wellformedness requirements of feature models. A feature model evolution plan, also called an evolution plan, can be presented as a sequence of feature models, each associates with a time point. An evolution plan without paradoxes is sound [14].*

1.2 A Motivating Example

We exemplify our merger on an example of a vending machine software product line. The evolution plans we present capture the planned changes of common vending machine features, such as different kinds of beverages, different types of currency, different sizes of cups, etc. The input to our merger contains a base evolution plan (shown in Fig. 2), and two new evolution plans, each a modified version of the base one.

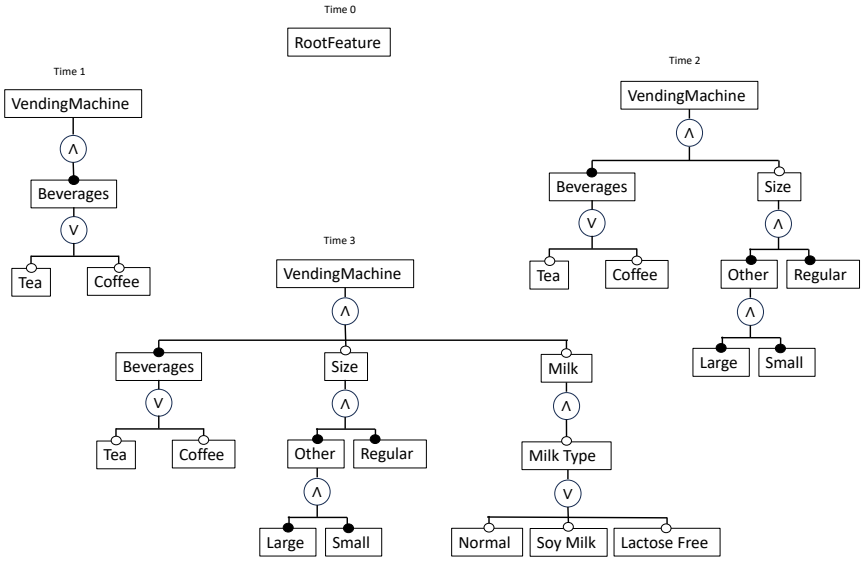

Fig. 2. Vending machine - base evolution plan

The base evolution plan in Fig. 2 consists of four distinct time points. Each time point is associated with a feature model and represents a milestone in the development of the vending machine. At Time 1, we introduce a mandatory Beverages feature, which means a vending machine has to provide the users an option of a beverage. Due to its *or*-group, a vending machine can choose if it wants to provide tea, coffee, none, or both. At Time 2, a collection of features and groups are added which provides options for cup sizes of beverages. The Size feature is optional, which means a vending machine does not have to provide different sizes of cups. If the Size feature is chosen, the vending machine has to supply large, small and regular cups. At Time 3, the Milk feature is introduced. If a vending machine chooses to provide milk, it can choose to provide zero or more of the features such as Normal, Soy Milk or Lactose Free. The base evolution plan represented the original planned development of the vending machine software product line. Since requirements often change, the plan can change. The changed plan is encoded as a completely new sequence of FMs. Due to the limitation of space, we describe how the updated evolution plans look in text rather than visualizing them in this paper.

The version of contributor 1 with respect to the base version At Time 1, the feature model looks the same as the one in the base plan at Time 1, except the *and* group of VendingMachine contains a new mandatory feature Currency, which has an *alternative* group containing two optional features Dollar and Euro. At Time 2, the feature model looks the same as the one in the base plan at Time 2, except the *and* group of the Size feature now contains only Large and Regular. In addition, the *and* group of VendingMachine contains a new mandatory feature

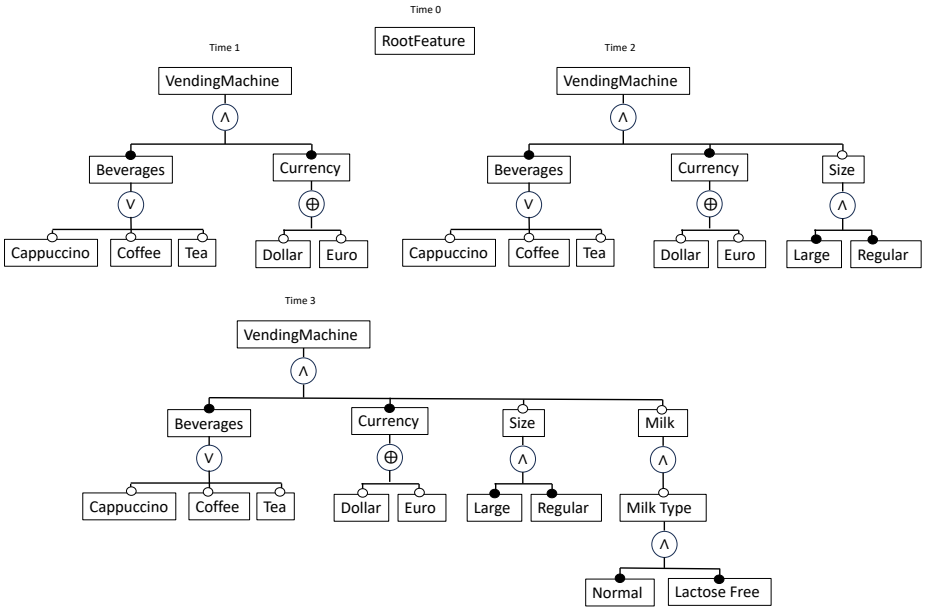

Fig. 3. Vending machine - the resulting evolution plan

Currency, which has an *alternative* group containing two optional features Dollar and Euro. At Time 3, the feature model looks the same as the one in the base plan at Time 3, except the *and* group of the Size feature now contains only Large and Regular, the Soy Milk feature is removed, the group type under the Milk Type feature is changed to *and*, and the feature types of Normal and Lactose Free are both changed to mandatory. Besides, the *and* group of VendingMachine contains a new mandatory feature Currency, which has an *alternative* group containing two optional features Dollar and Euro.

The version of contributor 2 with respect to the base version The feature models at Time 1∼3 look the same as the ones in the base plan at Time 1∼3, respectively, except they all have a new optional Cappuccino feature in their Beverages group. In addition, at Time 3, the group type under the Milk Type feature is changed to *and*, and the feature types of Normal and Lactose Free are both changed to mandatory.

The Resulting Evolution Plan Given these three evolution plans, our merger will detect the changes made in each of the new versions at each of the time point with respect to the base evolution plan and check if the changes are unifiable at each time point, i.e., if soundness of feature models is preserved after merging at each time point, respectively. In case paradoxes are detected, our merger will throw out error messages explaining the reason for failing and immediately stop the merging process. In this vending machine example, these three versions of evolution plans are unifiable. The final merged version is shown in Fig. 3.

2 A Soundness-Preserving Merger for Evolution Plans

In this section, we present the algorithm of our merger for evolution plans. The merge algorithm contains several distinct phases as illustrated in Fig. 4. Version 1 and version 2 represent inputs from two contributors, respectively. The algorithm contains the transformation of the evolution plan representations, as well as the detection of potential conflicts that could occur during the merging process. We will explain each phase of the algorithm one by one.

2.1 Phase 1: Construct an Intermediate Representation for Merging

The inputs to the algorithm are three sound evolution plans: the base evolution plan and two other re-planned versions from the collaborators. Each collaborator needs to ensure the soundness of their own evolution plan before merging so that we can leverage the soundness [14] of the input in the design of the algorithm. These three evolution plans inputted to the merger are defined as a sequence of tree-based FMs as described in Section 1.1. This tree structure works well for capturing the essence of evolution plans and is closest to what users would see and interact with. However, before merging the evolution plans, we need to calculate (1) the *modifications* between two subsequent FMs in an evolution plan, and (2) the *changes* between the evolution plans that will be merged at each time point. The tree-based FM requires recursively traversing two subsequent trees in an evolution plan to detect what modifications have been made between the two. It also makes the later stage of merging and unifying several plans challenging. A more suitable feature model representation is required.

The first phase of the merge algorithm, shown in Fig. 4, is performed by the `flattenEvolutionPlan` function, which converts the evolution plan from the TreeUserEvolutionPlan format to the FlatUserEvolutionPlan format. The representation of the transformed evolution plan is still a list of FMs. However, we transform the FMs from the tree structure to a map of features and a map of groups indexed by the feature IDs and group IDs, respectively. The mapping entry of a feature includes the feature name, feature type, and the ID of the group it belongs to. The mapping entry of a group includes the group name, group type, and the ID of the feature it belongs to. Since there is no parent-child hierarchy in a mapping, we say the FM structures are flattened. The flattened structure leverages the unique IDs to allow lookup based on IDs. Adding or removing a feature or group is rather simple and requires only adding or removing an entry in the mapping. Modifying the type of a feature or a group requires simply a lookup based on the ID, and then changing the fields of the corresponding mapping entry. Moving entire subtrees becomes straightforward with the use of mapping structures. It requires updating only the group field of the feature mapping entry.

2.2 Phase 2: Derive Modifications in a Plan

The goal of this phase of the algorithm is to make the *modifications* between subsequent FMs in an evolution plan explicit. The function `deriveModifications`

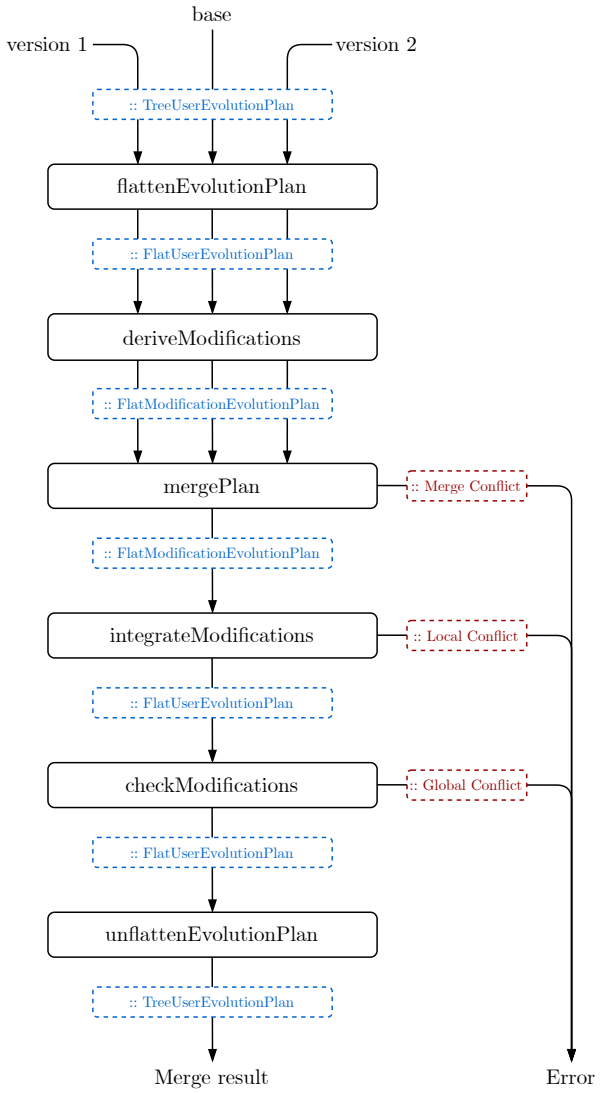

Fig. 4. Outline of the merge algorithm.

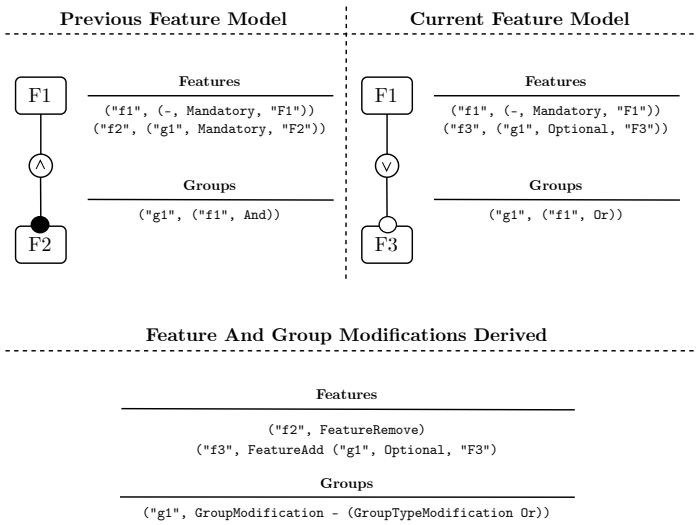

Fig. 5. The modifications between two subsequent FMs in an evolution plan.

transforms an evolution plan from the FlatUserEvolutionPlan representation to the FlatModificationEvolutionPlan representation, which keeps the initial FM but replaces each of the remaining FMs in the plan to a set of modifications necessary to transform the previous FM into the next one.

Fig. 5 shows an example of the modifications between two FMs. The top left represents the mapping of the previous FM and the top right represents the mapping of the current one. Compare the differences between the mappings, we can observe three modifications: (1) the mapping of feature `"f2"` is removed, (2) a new mapping for feature `"f3"` is added, and (3) the group type in the entry of mapping `"g1"` is modified from `And` to `Or`. The newly added feature mapping entry expresses that the feature named `"F3"` is added to group `"g1"`, and its feature type is optional. The bottom of Fig. 5 shows the three modifications necessary to transform from the previous FM into the current one. With the FlatModificationEvolutionPlan representation, the current FM on the top right corner of Fig. 5 will be replaced with the set of modifications listed at the bottom.

2.3 Phase 3: Detect Changes Between Versions

Before attempting to merge the new versions into the base version, we have to detect what *changes* have been made in each of the new versions with respect to the base one at each time point. In order to explain it clearly, let us highlight the difference between *modifications* and *changes* in this context again. With modifications, we refer to the differences between two subsequent FMs in an evolution plan as discussed in Section 2.2. The modifications are a part of the

evolution plan and have nothing to do with the changes between different versions. However, changes in a new evolution plan represent how the base evolution plan should be revised. Changes are not about adding, removing, or changing features and groups, but rather adding, removing, or changing the modifications. These changes are performed at the meta-level, allowing us to represent *changes to the modifications*. This distinction is subtle yet an important factor. If one of the new versions tries to remove a feature at a time point that did not exist in the modifications of the base version, this is represented as a change added to the plan. This is because the feature removal is represented as a new modification in the evolution plan, and we define this as "add" a new modification, which is the feature removal.

In some cases, one of the new versions might introduce new time points. To handle this, we collect the time points from all three inputted evolution plans and add an empty list of modifications to each time point that was absent in a given evolution plan so that the merging can be performed per time point.

The result of detecting the changes may have three different outcomes: There might be **No Change**, i.e., a modification existed in the base version and was not changed or removed in either version. It can be **Change In One**, i.e., a modification changed in one of the new versions. This includes three scenarios: (1) a modification did not exist in the base and was added in the new version, (2) a modification existed in the base but was removed in the new version, and (3) a modification from the base was changed to another modification in the new version. It can also be **Change In Both**, i.e., a modification changed in both versions. Similar to Changed In One, this includes changes where a modification existed in the base and modifications that did not exist in the base.

To clarify the concept of detecting changes between versions, we present an example in Fig. 6. *Version 1 Evolution Plan* includes two changes to the base evolution plan: (1) at Time 1, add an optional Feature 4 to the *and* group, (2) at Time 2, there was originally scheduled a modification of group type from *and* to *or*, but this version revises the modification to create an *alternative* group instead. *Version 2 Evolution Plan* includes only one change to the base evolution plan. At Time 2, it was originally scheduled a renaming of the root feature. However, in this version, the scheduled renaming is removed.

2.4 Phase 4: Merge Intended Changes

Once the changes have been detected between versions, the base evolution plan is ready to be integrated with all the changes from version 1 and version 2, if soundness is preserved. While merging the evolution plans, it is important to ensure soundness at each time point, i.e., the merged plan should follow the wellformedness requirements of an evolution plan. Even though all three given evolution plans are sound, merging all changes from different versions might still yield various conflicts, i.e., paradoxes. This phase of the algorithm is performed by the `mergePlan`, `integrateModifications`, and `checkModifications` functions. These three functions together ensure the soundness of a merged evolution plan by detecting any existing `Merge Conflicts`, `Local Conflicts`, and

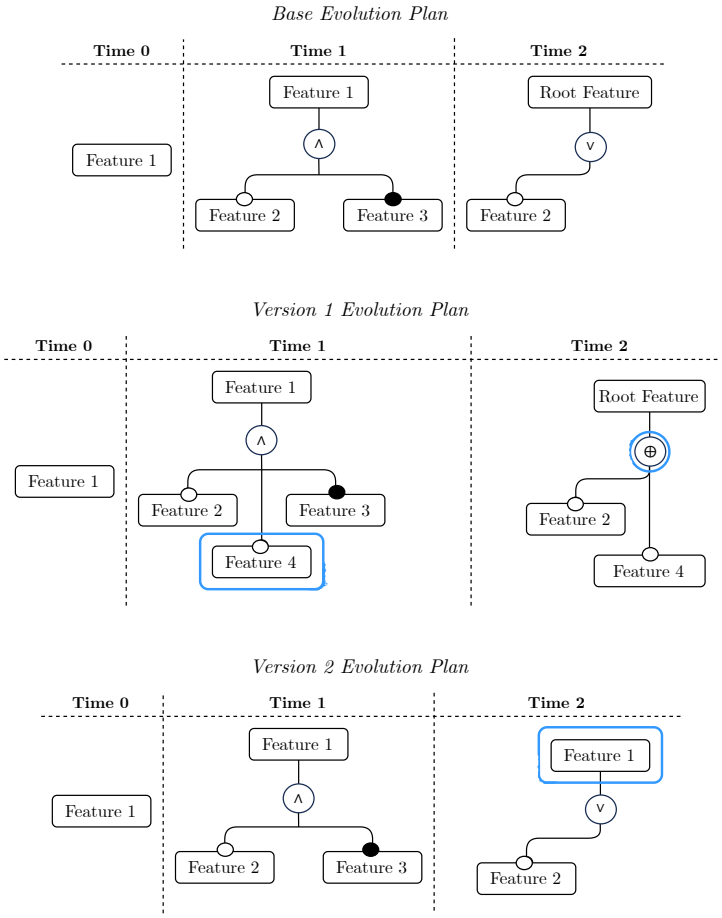

Fig. 6. An example highlights the changes in Version 1 and Version 2 with respect to the Base Evolution Plan in blue.

`Global Conflicts` in the merged plan. Note that there might be several changes of modifications from different versions at each time point. Some conflicts can be detected immediately and others can only be detected once all the changes of modifications belonging to the same time point have been applied. We will discuss them in detail below.

Detecting Merge Conflicts A `merge conflict` is raised due to diverging changes for a specific feature or group. For example, a `merge conflict` could happen if one version tries to remove a feature, while the other tries to change the type of a feature. It could also happen if there originally existed a modification in the base version, and one of the derived versions tries to change the modification, while the other tries to remove the modification. The merge conflicts can thus

be reported immediately without checking the rest of the modifications at the time point.

Detecting Local Conflicts `Local conflicts` occur when we try to alter or remove features or groups that *do not exist*, or when we try to add features that already *exist* at the same time points in different evolution plans. The local conflicts can thus be reported immediately without checking the rest of the modifications at the time point.

Detecting Global Conflicts While both merge conflicts and local conflicts are specific to the single feature or group at hand and can be raised without knowing what the rest of the modifications at the same time point are, some of the modifications rely on knowing the state of other features or groups, which prevents us from reporting these kinds of conflicts immediately. For example, adding features to parents that do not exist, removing groups that have children, or moving features so cycles are formed. Other violations of well-formedness can be if we change the type of a feature to something incompatible with its group type. However, detecting such `global conflicts` relies on we have already applied all the remaining modifications that belong to the same time point. For this reason, we postpone the soundness checking until after every modification at a time point has been included. Each modification yields a set of dependencies depending on the potential conflicts (Table 1). We generate a list of affected constraints we have to check for a given time point. A global conflict is raised if one or more affected constraints are not met.

When merging fails, our tool implementation provides information specific to the conflict, including the time point for the occurrence and the cause of the failing. A merged evolution plan from three sound versions remains sound if no conflicts are detected by the merger.

2.5 Phase 5: Unflatten the Representation

If no conflicts are detected, a successfully merged evolution plan will be the output of the computation. The very last phase of the merge algorithm executes the `unflattenEvolutionPlan` function, which converts the flattened FMs back to the original tree-based representation for visualization. The final merged evolution plan for the example in Fig. 6 is presented in Fig. 7.

3 Related Work and Conclusion

Several approaches and tools for planning the evolution of an SPL exist to this day, but none of them investigated merging strategies for different versions of updated evolution plans. DeltaEcore is a tool suite relying on Hyper-Feature Models (HFMs) and evolution delta modules for the integrated management of variability in space and time in SPLs [25]. DarwinSPL is a tool suite for modeling evolving context-sensitive SPLs by extending DeltaEcore's HFMs [20]. EvoFM

Modification Type	Affected Constraints
Add Feature	ParentGroupExists UniqueName FeatureIsWellFormed
Remove Feature	NoChildGroups
Modify Feature Parent	ParentGroupExists NoCycleFromFeature FeatureIsWellFormed
Modify Feature Type	FeatureIsWellFormed
Modify Feature Name	UniqueName
Add Group	ParentFeatureExists
Remove Group	NoChildFeatures
Modify Group Parent	ParentFeatureExists NoCycleFromGroup
Modify Group Type	GroupIsWellFormed

Table 1. Affected constraints

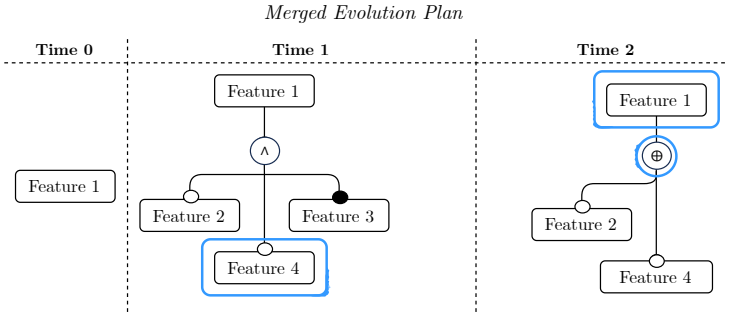

Fig. 7. A visualization of the merged result

supports long-term feature-oriented planning and analysis of evolution plans [7]. FORCE [13], Feature-Driven Versioning [19], and SuperMod [24] also support evolution planning of SPLs. A term rewriting system for soundness checking for evolution plans was introduced and integrated in [14].

Model-driven engineering (MDE) and graphical domain-specific languages are related research areas. In MDE, models are the primary artifacts of the software development process. There exists abundant work on model evolution and version control. In the survey [26] one can find several model comparison approaches and applications. Model differencing include calculation of matching model elements, representation of differences, and visualization of the differences. Works that describe how to compute the difference of models include EMF Compare [12], DSMDiff [17], and [18] just to mention a few. Difference calculation is often divided into phases: to detect model mappings, where all the elements of the two input models are compared through metrics that are static identity-based, signature-based, similarity-based [12], or language-specific [29], and to de-

termine model differences, where all the additions, deletions and changes are detected. Kolovos et al. [16] survey approaches for model matching. Several repositories of version control systems for models have been proposed. Survey [4] gives state-of-the-art versioning systems dedicated to modeling artifacts and some solutions also provide resolutions for model conflicts, e.g., in [8]. EMFStore [15] and CDO [1] support collaborative editing and versioning of models and are model repositories for EMF. Other commercial modeling tools include [2,3]. What makes our work immediately different is that the objects to be merged are evolution plans and not single models. The version control system needs to take into account the temporal aspect and work with multiple models that are involved when planning evolution.

This work introduced a soundness-preserving merger for feature model evolution plans. We leverage the soundness of the input in the design of the algorithm: the base evolution plan and two other re-planned versions from the contributors are given to the algorithm. These plans are merged and soundness is preserved in the resulting plan if no conflict exists. A tool implementation based on the described algorithm has been implemented in Haskell and evolution plans can be visualized [27,28]. We believe such a merge tool can give engineers a better toolbox for dealing with the parallel development of evolution plans. We leave the evaluation of the algorithm and its application to real-world scenarios for future work.

References

1. CDO Model Repository. `https://eclipse.dev/cdo/`.
2. LabView. `https://www.ni.com/en-us/support/documentation/supplemental/21/managing-labview-vi-and-application-revision-history.html`.
3. MetaEdit+. `https://www.metacase.com/news/smart_model_versioning.html`.
4. K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *Int. J. Web Inf. Syst.*, 5:271–304, 2009.
5. V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. J. P. de Lucena. Refactoring product lines. In *Generative Programming and Component Engineering, 5th Int. Conf.*, pages 201–210. ACM, 2006.
6. D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines, 9th Int. Conf., SPLC 2005, Rennes, France, Proc.*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
7. G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. EvoFM: feature-driven planning of product-line evolution. In *Proc. 2010 ICSE Workshop Product Line Approaches in Software Engineering*, pages 24–31. ACM, 2010.
8. P. Brosch, M. Seidl, and G. Kappel. A Recommender for Conflict Resolution Support in Optimistic Model Versioning. New York, NY, USA, 2010. Assoc. for Computing Machinery.
9. J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. Reasoning about Product-Line Evolution using Complex Feature Model Differences. In *Software Engineering 2017, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume P-267 of *LNI*, pages 67–68. GI, 2017.
10. P. Clements and L. M. Northrop. *Software product lines - practices and patterns*. SEI series in software engineering. Addison-Wesley, 2002.

11. O. Díaz, L. Montalvillo, R. Medeiros, M. Azanza, and T. Fogdal. Visualizing the customization endeavor in product-based-evolving software product lines: a case of action design research. *Empir. Softw. Eng.*, 27(3):75, 2022.
12. Eclipse. EMF Compare. `https://eclipse.dev/emf/compare/`.
13. D. Hinterreiter, H. Prähofer, L. Linsbauer, P. Grünbacher, F. Reisinger, and A. Egyed. Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems. In *23rd IEEE Int. Conf. on Emerging Technologies and Factory Automation*, pages 107–114. IEEE, 2018.
14. A. Hoff, M. Nieke, C. Seidl, E. H. Sæther, I. S. Motzfeldt, C. C. Din, I. C. Yu, and I. Schaefer. Consistency-preserving evolution planning on feature models. In *24th ACM Int. Systems and Software Product Line Conf.*, pages 8:1–8:12. ACM, 2020.
15. M. Koegel and J. Helming. EMFStore: A Model Repository for EMF Models. New York, NY, USA, 2010. Assoc. for Computing Machinery.
16. D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. USA, 2009. IEEE Computer Society.
17. Y. Lin, J. Gray, and F. Jouault. DSMDiff: A differentiation tool for domain-specific models. *European Journal of Information Systems*, 16:349–361, 08 2007.
18. S. Maoz and J. O. Ringert. A Framework for Relating Syntactic and Semantic Model Differences. 17(3), 2018.
19. R. Mitschke and M. Eichberg. Supporting the Evolution of Software Product Lines. In *ECMDA Traceability Workshop*, 2008.
20. M. Nieke, G. Engel, and C. Seidl. DarwinSPL: an integrated tool suite for modeling evolving context-aware software product lines. In *Proc. 11th Int. Workshop Variability Modelling of Software-intensive Systems*, pages 92–99. ACM, 2017.
21. A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven support for product line evolution on feature level. *J. Syst. Softw.*, 85(10):2261–2274, 2012.
22. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
23. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *Int. J. Softw. Tools Technol. Transf.*, 14(5):477–495, 2012.
24. F. Schwägerl. *Version Control and Product Lines in Model-Driven Software Engineering*. PhD thesis, University of Bayreuth, Germany, 2018.
25. C. Seidl, I. Schaefer, and U. Aßmann. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Modellierung 2014, Wien, Österreich*, volume P-225 of *LNI*, pages 81–96. GI, 2014.
26. M. Stephan and J. R. Cordy. A Survey of Model Comparison Approaches and Applications. In *Proc. 1st Int. Conf. Model-Driven Engineering and Software Development*, pages 265–277. SciTePress, 2013.
27. E. H. Sæther. Three-Way Semantic Merge for Feature Model Evolution Plans. Master's thesis, University of Oslo, 2021. `http://urn.nb.no/URN:NBN:no-89666`.
28. E. H. Sæther, I. C. Yu, and C. C. Din. Software artefact for: "Semantics-Based Version Control for Feature Model Evolution Plans", Sæther et al., NIK 2023, 2023. `https://doi.org/10.5281/zenodo.10044858`, original source at: `https://github.com/eirikhalvard/master-thesis`.
29. Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. New York, NY, USA, 2005. Assoc. for Computing Machinery.