Evolving Delta-Oriented Product Lines: A Case Study on Feature Interaction, Safe and Partially Safe Evolution

Leomar Camargo, Luisa Fantin, Gabriel Lobão
Thiago Figueiredo, Rodrigo Bonifácio
Computer Science Department, University of Brasília
Brasília, Brazil

Karine Gomes
Leopoldo Teixeira
Informatics Center, Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Software product line engineering is a well-known approach for building a set of configurable systems for a specific domain, and different techniques have been used to manage product line variability, including source-code preprocessing, aspect-oriented programming (AOP), and delta-oriented programming (DOP). Although existing studies have explored the design and evolution of product lines using techniques such as source-code preprocessing and AOP, little is known about the practical implications of using DOP to bootstrap and evolve software product lines. In this paper we address this issue, reporting our experience of using DeltaJ to implement two product lines (Reminder-PL and Iris-PL). This experience covers different scenarios of evolution (such as the inclusion of mandatory, optional, and alternative features) that indeed led to several feature interactions. Altogether, this work brings several contributions, including evidence that existing templates for safe and partially safe evolution of product lines can also help developers to evolve delta-oriented SPLs—although we revealed the need for two additional templates for safe evolution. Also, we present a description of the feature interactions that appeared during the evolution of both product lines and how we modularized these interactions using DOP constructs.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Software evolution**; • **General and reference** → *Empirical studies.*

KEYWORDS

Delta-Oriented Programming, Software Product Lines (SPLs), Safe and Partially Safe Evolution of SPLs, Feature Interaction

ACM Reference Format:
Leomar Camargo, Luisa Fantin, Gabriel Lobão, Thiago Figueiredo, Rodrigo Bonifácio, Karine Gomes, and Leopoldo Teixeira. 2021. Evolving Delta-Oriented Product Lines: A Case Study on Feature Interaction, Safe and Partially Safe Evolution. In *Brazilian Symposium on Software Engineering (SBES '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474624.3474645

SBES '21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9061-3/21/09…$15.00
https://doi.org/10.1145/3474624.3474645

1 INTRODUCTION

Software product lines (SPLs) are an efficient way of developing sets of software products with similar functionality and behaviors [30]. This approach can increase developers' productivity while reducing costs and development time—due to the possibility of reusing a common set of assets [14]. Nonetheless, SPLs also require maintenance effort, which might involve bug fixes and introducing new features and functionalities. As a result, SPLs constantly evolve by adding, removing, or changing features and SPL assets. This can be a costly and tedious process that might introduce defects and unintended modifications to the behavior and functionalities of a variety of products—nullifying previous productivity gain, cost and time savings [28].

Some introduced defects might be particularly difficult to detect because they are present only in certain products, involving specific feature interactions. This might require a large number of tests to discover and fix the problem. As the number of products increase, testing all product variants becomes increasingly expensive and unproductive [36]. In order to mitigate this problem, previous studies have proposed template catalogs for the safe and partially safe evolution of SPLs [10, 12, 28, 32]. Although these catalogs have been validated using compositional and annotative SPLs, it is still unclear whether or not these catalogs suffice to guide developers using transformational techniques such as Delta-Oriented Programming (DOP) [33]—a prominent approach for managing SPL variability. Besides that, to the best of our knowledge, there is no discussion in the literature about how to modularize feature interactions using DOP. In this paper we seek to address these issues by discussing an experience report on using existing safe and partially safe evolution catalogs to evolve two DOP product lines and showing recurrent patterns of code we used to modularize feature interactions using delta-oriented constructs.

The rationale for this research is two fold. First, regarding SPL evolution, it is important to characterize how changes were performed into existing projects, and how they relate to safe and partially safe evolution notions. This characterization might result in developing better tools to assist developers in performing such changes and assist SPL developers in the challenging task of evolving product lines. Our work provides evidence that some SPL evolution operations can be expressed as templates, regardless of the variability implementation mechanism. Although we give evidence that the catalogs could also be used for transformational techniques (DOP), we revealed the need for additional templates in this work and presented a mapping of the SPL concepts used in the catalogs into the DeltaJ constructs.

Second, existing literature about DOP claims that it is possible to modularize every feature in an independent delta-module. Our

experience suggests the contrary, and as we discuss in this paper, it was often necessary to use a delta-module to modularize feature interactions in our case studies. For this reason, here we also report recurrent code idioms that emerged in our case studies. The code idioms could be reused by other researchers and developers interested in DOP. In summary, our work makes the following contributions:

- We provide new evidence that existing catalogs for safe (and partially safe) evolution of SPLs can assist developers during the evolution of delta-oriented product lines.
- We describe how to manage feature interactions using DOP constructs using a set of scenarios. In addition, we suggest a few code idioms to modularize feature-interactions using DOP.
- We implement two full-fledged delta-oriented SPLs (Reminder-PL and Iris-PL) and make these product lines available for further studies. [1, 2]

2 BACKGROUND

In this section, we provide an overview of foundational concepts for understanding our work. For our purposes, an SPL is formally represented as three elements: *Feature Model* (FM), which describes the variability of an SPL; *Asset Mapping* (AM), representing the SPL core assets; and *Configuration Knowledge* (CK), which maps *feature* expressions to functions that might select or transform core assets [12]. Feature expressions are boolean formulae denoting presence conditions among features. That is, a CK entry might be in the form $(f_1 \wedge (f_2 \vee f_3)) \mapsto select(n_1)$, stating that if a product is configured with the feature f_1 and either f_2 or f_3 (or both), we must select the asset named n_1 during the build process.

2.1 Delta-oriented Programming

Delta-oriented Programming is a transformational technique for variability management in SPLs [33]. Although it follows the same principle of gradual development from Feature-oriented Programming [31], its design weakens a specific FOP constraint. That is, besides operations for adding and modifying modules, DOP also supports operations for removing existing modules. In this way, DOP corresponds to a more expressive and flexible approach for SPL development when compared to FOP.

In its earlier versions, DOP ensured a separation between the core module and a set of delta modules. The core module could be understood as a valid SPL instance implementation, according to the FM constraints. This instance should contain at least the mandatory features and a minimum set of optional features, according to some design considerations. Delta modules specify which changes should be applied to the core module to implement new products, by *adding*, *changing*, or *removing* classes, attributes, or methods.

The core ideas of DOP have been used to manage variability not only at the source code level [1, 15, 38], but also in different artifact types [24, 26]. Here we use a specific implementation of DOP for Java (DeltaJ 1.5 [23]), to report the design, implementation, and evolution of two SPLs: an Android product line application (Reminder-PL) and a standalone Java e-mail client (Iris-PL).

[1] https://github.com/Reminder-App
[2] https://github.com/iris-email-client

Listing 1 shows a simplified version of the DeltaJ configuration file for Reminder-PL, one of the SPLs we use in our research. This file maps SPL concepts to DeltaJ implementation constructs.

```
1  SPL ReminderPL {
2     Features = {Reminder, ReminderCategory, StaticCategory,
3       ManagedCategory, Priority}
4
5     Deltas = {dBase, dStaticCategory, dManagedCategory, dPriority,
6       dStaticCategoryPriority}
7
8     Constraints {
9        Reminder & ReminderCategory | Priority;
10       ReminderCategory & (StaticCategory ^ ManagedCategory);
11    }
12
13    Partitions {
14       {dBase} when (Reminder);
15       {dStaticCategory} when (StaticCategory);
16       {dManagedCategory} when (ManagedCategory);
17       {dPriority} when (Priority);
18       {dStaticCategoryPriority} when (StaticCategory & Priority);
19    }
20
21    Products {
22      Prod1 = {Reminder, ReminderCategory, StaticCategory, Priority};
23      Prod2 = {Reminder, ReminderCategory, ManagedCategory};
24    }
25 }
```

Listing 1: Example of a DeltaJ configuration file.

Considering a DeltaJ configuration file for a product line (such as that we present in Listing 1), we associate the feature model (FM) to the `Features` and `Constraints` sections. The first section (`Features`) determines the set of all features in the SPL—in the example, we have five features: *Reminder*, *ReminderCategory*, *StaticCategory*, *ManagedCategory*, and *Priority*. The `Constraints` section states the feature model constraints by means of propositional formulae. Line 7 in the `Constraints` section states that *Reminder* and *ReminderCategory* are mandatory, while *Priority* is an optional feature, and line 8 states that *ReminderCategory* requires the selection of either *StaticCategory* or *ManagedCategory*, but not both.

The `Deltas` section lists the names of the delta modules used to implement the SPL features. As such, it corresponds to the AM (*asset mapping*) element of the formal SPL definition. Finally, the `Partitions` section contains the mapping between features and delta modules, and thus corresponds to the CK element. In the example, the *Reminder* feature requires the `dBase` delta module, the *StaticCategory* feature requires the `dStaticCategory` delta module, and so on. Finally, the `Products` section lists the products that one might be interested in building from the SPL. Listing 1 illustrates just two products, although other products could be generated from this DeltaJ definition.

2.2 Safe Evolution

Like any software system, SPLs also evolve. However, evolution must be taken into account carefully in SPLs, as simple changes can impact the behavior of several products. This issue motivated previous research to investigate new methods to help developers minimize the impact of SPL evolution [27, 29]. For instance, the concept of *Safe Evolution* of SPLs argues in favor of *behavior preserving transformations* that might help to "safely evolve" an SPL [12]. *Safe evolution* here means that, after evolving the SPL, we should be able to produce at least the same set of products of the previous version. Previous works proposed *templates* for SPL transformations that

reportedly guarantee *safe evolution* [12, 28, 35], giving orientation on how to safely evolve an SPL.

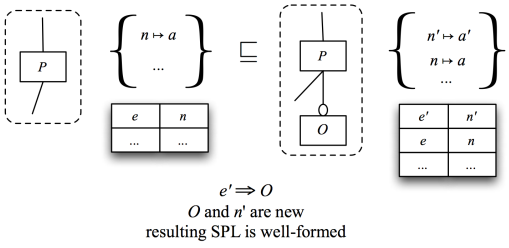

Figure 1: Add New Optional Feature template.

Figure 1 presents the Add New Optional Feature transformation template [12, 28]. The safe evolution templates consist of a left-hand side (SPL before the transformation) and a right-hand side (SPL after the transformation). In both sides, it shows fragments of the SPL elements (i.e., FM, AM, and CK). For instance, in the left-hand side of Figure 1, we show a feature model fragment with a sole feature *P*; a fragment of an asset mapping, mapping the name *n* to the artifact *a*; and a configuration knowledge fragment relating the feature expression *e* to the artifact name *n*. In the right-hand side, we enrich the feature model with an optional feature *O* and introduce a new mapping ($n' \mapsto a'$) and a new entry in the configuration knowledge (feature expression *e'* relating to *n'*).

The template in Figure 1 establishes that we can *safely add* a new optional feature *O* to an SPL that matches the left-hand side, together with a new asset *a'*, if this asset is only included when we select the new feature. This motivates the restriction of establishing that the feature expression *e'* evaluates to true when *O* is selected ($e' \Rightarrow O$). This guarantees that products without the new feature correspond exactly to their original counterparts. The template also requires that we cannot have a feature named *O* in the FM nor another asset name *n'* in the AM on the original SPL, and that the resulting SPL is well-formed. Safe evolution is guaranteed because the resulting SPL generates all products that it had before plus the new products that contain feature *O*, and we improve the resulting SPL quality by increasing its configurability. The template formalization allows associating more than one asset to the new optional feature in the CK.

2.3 Partially Safe Evolution

Making safe modifications allows developers to evolve SPLs with basic guarantees that existing products would not be affected by the evolution scenario. However, during the life cycle of a software project, often we need to remove features, fix bugs, or change the implementation. These kinds of changes affect the behavior of existing products and thus are not covered by the safe evolution notion. Therefore, Partially Safe Evolution [32] considers changes that preserve the behavior of *some, but not all existing products*. Sampaio et al. also suggested transformation templates that abstract common scenarios that impact existing instances of an SPL [32]. Here, we leverage existing catalogs to report on both safe and partially safe evolution of delta-oriented product lines [28, 32, 35].

3 STUDY SETTINGS

The objective of this study is to (a) explore the use of existing templates to guide the safe and partially safe evolution of delta-oriented SPLs; and to (b) report our experience on evolving and modularizing feature interactions in delta-oriented SPLs. We conduct our investigation using two SPLs: Reminder-PL and Iris-PL (see Section 3.1). Altogether, we seek to answer the following research questions.

- (RQ1) How existing safe and partially safe evolution templates have been used to support the evolution of Reminder-PL and Iris-PL?
- (RQ2) Which code idioms have been used to modularize feature interactions during the evolution of Reminder-PL and Iris-PL?

The first research question aims to characterize the evolution of delta-oriented SPLs using existing safe and partially safe evolution templates. Exploring this research question allows us to understand if these templates are useful to assist developers on evolving SPLs that use transformational techniques, such as DOP. For RQ2, the purpose is to analyze the feature interactions among delta-oriented SPLs. This allows us to understand which idioms one might use to modularize feature interactions using delta-oriented constructs.

3.1 Case Studies

We answer our research questions using two SPLs (Reminder-PL and Iris-PL), which were extracted from individual products and then evolved according to a set of evolution scenarios. All data related to this study, including the SPL artifacts, can be found in our online appendix.[3]

3.1.1 Reminder-PL. Reminder App is a mobile application developed for Android devices using Java 1.5 and the Android SDK 16 (or Android 4.1). We first developed a full-fledged Android application in cooperation with an industrial partner. That is, the original version of the app was not designed with the purpose of conducting research neither in DOP nor in SPLs.

In its original version, Reminder App supports the management of *reminders* (registration, editing, viewing, and deletion), allowing a reminder to be shared in Google Calendar. During the process of registering a reminder, it is possible to define a Priority ("No Priority", "Important", or "Urgent") and a Category ("Personal", "Work", or "College") for the reminder. It is also possible to add, edit, and delete reminder categories.

We chose Reminder App for several reasons. First, we had developed the first official releases of this application. Second, this application presents several opportunities for introducing variability (for instance, the support for Google Calendar could be made optional). Third, to the best of our knowledge, there are no reports on implementing Android SPLs using delta-oriented constructs.

The process we use for transforming Reminder App into a SPL involved three distinct phases:

- Domain engineering: where we decided the variability space for Reminder-PL.
- Converting Reminder App into an SPL: where we reduced the number of implemented features to generate a set of core modules using DeltaJ. The "core modules" should not be

[3] https://github.com/leomarcamargo/sbes-2021-package

affected by introducing new features in the next releases, and consists of the first release of Reminder-PL.
- New features development: where we used an incremental approach to evolve Reminder-PL, particularly introducing new features and refactoring the design to minimize code duplication.

Figure 2 shows the Reminder-PL FM. We can see that four *releases* were implemented during the SPL evolution process. The final implementation generates up to 60 valid products—the total number of configurations possible according to the restrictions in the FM. Table 1 presents some figures corresponding to the size of Reminder-PL in DOP considering the number of components and LOC. For components, we consider Java classes (in the case of the Base release), delta modules, and XML files used to implement the GUI of the app.

	BASE	v1	v2	v3	v4
Components	72	41	53	72	77
LOC	3427	2629	4198	6314	7425

Table 1: Metrics regarding the size of Reminder-PL.

3.1.2 Iris-PL. Unlike Reminder App, the Iris E-mail Client was initially designed to conduct research on feature interaction and SPL engineering using DOP. We first implemented a version of Iris using Java, in order to build a resilient architecture around object-oriented design patterns. The rationale was to have an architecture that would facilitate introducing new *e-mail client commands*—on top of an initial command line interface. After that, we migrated all features of this first version to DeltaJ, and started to implement new releases, adding new optional, alternative, and OR features using delta-oriented constructs. This SPL supports many functionalities often found in email clients, including sending, receiving, and forwarding messages; message tagging; message encryption; persistence; and searching mechanisms. We decided to implement this SPL because configurable email clients have been well discussed in the literature, and seminal works use email clients to illustrate feature interaction problems [8, 19]. Figure 3 shows the Iris-PL FM and Table 2 presents some metrics regarding the size of Iris-PL. First in the base implementation (Java) and then in the different DOP releases (v1 – v4). The table also shows the number of components (Java classes or delta modules) and lines of code (LOC) in each *release*.

	BASE	v1	v2	v3	v4
Components	71	23	27	39	43
LOC	2782	4230	4372	5185	5533

Table 2: Metrics regarding the size of Iris-PL.

3.2 Analyzing The Use of Existing Templates to Characterize Safe Evolution

To answer RQ1, we consider the evolution history of each SPL release, with the intent to ascertain if the existing safe and partially

```
Reminder: ManageReminder[1] GUI[1] ReminderSchedule
        [ReminderCategory][2] [Search][3] [GoogleCalendar][3]
        [ReminderPriotity][3];
ManageReminder: Create Edit View Delete Done;
ReminderSchedule: FixedDate[1] | DateRange[3] | DateRepeat[4];
ReminderCategory: ManagedCategory | StaticCategory;

DateRepeat IMPLIES ~GoogleCalendar;
```

Figure 2: Reminder-PL feature-model using a grammar notation [9]. In this notation, features in *brackets* are optional, and alternatives to a feature appears using a | symbol. The numbers in each feature represent the release in which the feature was implemented.

```
Iris: Services[1] UI[1] Persistence[1] [AddressBook][1] [Tagging][1]
    [Search][2] [Security][3] [Category][4];
Services: SendMessage ReceiveMessage;
UI: Console | GUI;
Persistence: Relational | NonRelational;
Search: SimpleSearch | AdvancedSearch;
Security: Sign ∨ Verify ∨ Encrypt ∨ Decrypt;

Sign IMPLIES Verify and Verify IMPLIES Sign;
Encrypt IMPLIES Decrypt and Decrypt IMPLIES Encrypt;
SimpleSearch IMPLIES Relational;
AdvancedSearch IMPLIES NonRelational;
```

Figure 3: Iris-PL feature-model using a grammar notation [9]. The numbers in each feature represent the release in which the feature was implemented.

safe evolution templates could characterize and help the evolution scenarios of the DOP implementations of Reminder-PL and Iris-PL. We analyze the commits to identify the evolution scenarios that match existing templates. Altogether, here we report the results of an assessment of 43 commits from Reminder-PL, and 115 commits from Iris-PL—these numbers correspond to all commits for both projects. This task was carried out by two authors that contributed to the Reminder-PL development and one author that contributed to the Iris-PL. Table 3 shows the total number of analyzed commits in each SPL release.

SPL	v1	v2	v3	v4
Reminder-PL	11	11	15	6
Iris-PL	79	21	13	2

Table 3: Amount of commits by release.

In our context, we consider that an evolution scenario (such as including a new feature or a bug fix) might involve a set of commits. We use the `Repodriller`[4] tool to collect information from each commit, and we analyze the content of each commit to manually group them. Working in pairs and inspecting the decisions in group, we associate the commits to the evolution scenarios that form a release. We then investigate, considering all evolution scenarios, the

[4]https://github.com/mauricioaniche/repodriller

impact on the SPL assets (i.e., FM, AM, CK), so that we could characterize a given scenario according to an existing safe or partially safe evolution template.

3.3 Understanding Feature Interaction Patterns in DOP Product Lines

For our purposes, a feature interaction occurs whenever the selection of a feature changes the behavior of another feature, as previously defined [6]. Here we only consider feature interactions involving at least a non-mandatory feature (a_1), which might change either the behavior of a *mandatory* feature (m_1) or another *non-mandatory* feature (a_2). In the first case, which we call *Type (a)*, we identify the occurrence of a feature interaction between a_1 and m_1 whenever an alternative feature a_1 requires a delta module $delta_1$, the mandatory feature m_1 requires a delta module $delta_2$ that declares a class c_1, and $delta_1$ needs to modify the c_1 class.

In the second case, namely *Type (b)*, we identify the occurrence of a feature interaction between two non-mandatory features a_1 and a_2 whenever we have an entry on the CK (more precisely, an entry on `partition` clause of a DeltaJ product line definition) stating that a product with both features ($a_1 \wedge a_2$) must include a specific delta module $delta_1$. This scenario usually occurs when it is necessary to have the two features present in a given product configuration, where the individual deltas of each feature are not sufficient for the resulting product. We manually analyzed the source code of all Reminder-PL and Iris-PL releases to compute the occurrences of both types of interactions. The next step was to identify which idioms have been used to modularize the source code needed to handle the feature interactions.

4 RESULTS AND DISCUSSION

In this section, we present and discuss the results of our investigation, that aims to answer our research questions RQ1 (*How existing safe and partially safe evolution templates have been used to support the evolution of Reminder-PL and Iris-PL?*) and RQ2 (*Which code idioms have been used to modularize feature interactions during the evolution of Reminder-PL and Iris-PL?*).

4.1 Answers to Research Question (RQ1): Use of the templates for safe and partially safe evolution

We investigated whether or not developers might benefit from existing catalogs of safe and partially safe evolution templates in the process of evolving DOP SPLs. To this end, we reviewed the contributions necessary to evolve both SPLs across the different releases, and mapped such contributions to the templates, whenever we found a correspondence.

4.1.1 Reminder-PL results. The main goal of the second release of Reminder-PL (v2) was to introduce a new optional feature Category, which supports two variants: Static Category and Managed Category. The Managed Category feature allows Reminder-PL users to define their own set of categories for classifying reminders. To support this scenario, we applied a composition of two templates: in the first, we introduced an optional feature Static Category using the Add New Optional Feature template and then we created an alternative group Category with two variants—using a new template (Change Optional to Alternative Feature) that we designed for this research. Listing 2 illustrates this evolution scenario (consider that the `Reminder` feature represents all mandatory features of Reminder-PL).

```
SPL ReminderPL {
  Features = {Reminder, StaticCategory, ReminderCategory,
              ManagedCategory}
  Deltas = {dReminder, dStaticCategory, dManagedCategory}
  Constraints {
     Reminder | ReminderCategory & (StaticCategory ∧ ManagedCategory);
  }
  Partitions {
     {dReminder} when (Reminder);
     {dStaticCategory} when (StaticCategory);
     {dManageReminder} when (ManagerReminder);
  }
  Products {
    // ...
  }
}
```

Listing 2: Adding `Manage Category` feature.

When inserting the Managed Category feature, it was necessary to add an abstract feature Reminder Category to group its respective alternative features. In addition, the Static Category feature, which was previously an optional feature, became an alternative feature from this point of evolution. After analyzing the safe and partially safe evolution template catalogs, we did not identify a template that described the change from an optional feature to an alternative feature and therefore we proposed a new template, as described in Figure 4.

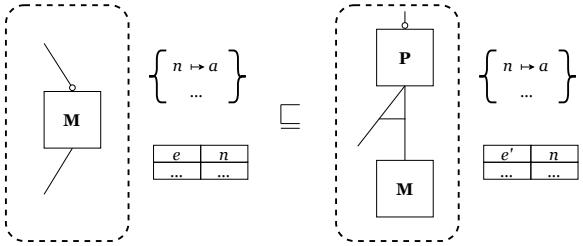

Figure 4: Change Optional to Alternative Feature template.

Considering this template, it is possible to transform an optional feature M in an alternative feature by associating this feature to a new feature expression e' only if the constraint that says selecting e' implies selecting M is satisfied. The second constraint states that this transformation is only possible if the new expression e' only contain feature names that already belong to the FM. Finally, the last constraint says that P must be an optional feature whose name does not exist in the FM.

Regarding the third release of Reminder-PL (v3), its main goal was to introduce four additional features, including the Date Range feature that allows the registration of a reminder with a start and an end date. In the previous release, a reminder could only be created for a specific day. This inclusion caused a change in the initial schedule structure of the application, so that the Fixed Date

feature—that was previously mandatory—became an alternative feature, with Reminder Schedule as an abstract (and mandatory) feature and Fixed Date and Date Range as its alternatives. For this evolution scenario, we used a new template (Change Mandatory to Alternative Feature) that we represent in Figure 5. This evolution is similar to the one presented in Listing 2, even though it uses different features.

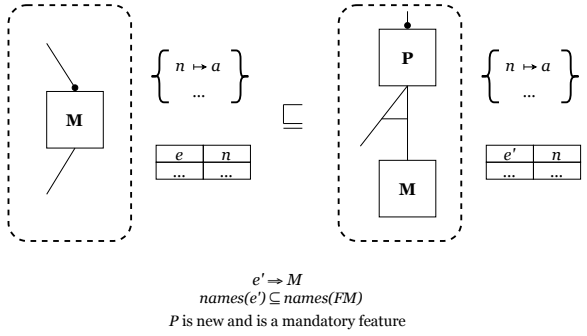

Figure 5: Change Mandatory to Alternative Feature template.

In the new template Change Mandatory to Alternative Feature (Figure 5), a mandatory feature *M* is converted into an alternative feature by associating this feature to a new feature expression *e'*—that is, selecting *e'* implies in selecting *M*. Similarly to the new template in Figure 4, this transformation is only possible if the new expression *e'* uses names from the FM. The last constraint regarding this template is that the new feature *P* must be mandatory. In addition, three new optional features were included in Reminder-PL v3, following the Add New Optional Feature template. These optional features include new options for customizing the application, such as adding priority to reminders (Reminder Priority feature) and sharing reminders using the Google calendar service (Google Calendar feature). We also fixed bugs in this release and improved the implementation of a feature using the Change Asset template. One of the bug fixes was related to the Manage Category feature.

Finally, the goal of the last release (v4) of Reminder-PL was to introduce an alternative feature (Date Range), in addition to fixing some bugs in the SPL. However, we could not classify the introduction of Date Range feature as safe—as we explain in Section 4.1.3. However, the bug fixes present in two commits followed the Change Asset template.

4.1.2 Iris-PL results. We also identified at least one template (either safe or partially safe) in each release of Iris-PL. The initial commits of the first release deal with the transformation of the Java single-sytem into a delta-oriented product line. This process does not fit in the safe evolution notion, which can only relate an existing SPL with its evolved version. This is why the existing templates do not cover the scenario of extracting SPLs from a single product. Therefore, from the moment this transformation was completed, we were able to analyze the SPL evolution according to each scenario. In Iris-PL we identified sets of commits that relate to more than one feature at the same time. For these cases, we computed the templates that characterized a given evolution scenario as a transaction that might involve different commits.

After transforming the original system into an SPL, the second release of Iris-PL introduces three features, two optional features (*Address Book* and *Tagging*) and an alternative: *Lucene DB*, which was later renamed to *Non Relational*, following the Add New Optional Feature and Add New Alternative Feature templates. In the third release, the goal was to introduce security-related features—decryption and encryption of messages while sending and receiving messages. For this, an optional feature, containing a group of *or-features*, has been added to the SPL, following the Add New Optional Feature and Add New OR Feature templates. Listing 3 illustrates this scenario. Consider the Service feature as corresponding to all mandatory features of Iris-PL.

```
SPL IrisPL {
   Features = {Service, Security, Encrypt, Decrypt, Sign, Verify}
   Deltas = {dService, dEncrypt, dDecrypt, dSign, dVerify}
   Constraints {
      Service | Security & (Encrypt ∨ Decrypt ∨ Sign ∨ Verify);
      (Encrypt IMPLIES Decrypt) & (Decrypt IMPLIES Encrypt);
      (Verify IMPLIES Sign) & (Sign IMPLIES Verify);
   }
   Partitions {
      {dService} when (Service);
      {dEncrypt} when (Security & Encript);
      {dDecrypt} when (Security & Decrypt);
      {dSign} when (Security & Sign);
      {dVerify} when (Security & Verify);
   }
   Products {
     // ...
   }
}
```

Listing 3: Adding `Security` feature.

In addition, the third release includes bugfixes and improvements for the *Address Book* feature—introduced in the first release of the SPL. Finally, the last release of Iris-PL, which has only two commits, aimed to introduce the optional feature *Category*, following the definitions in the Add New Optional Feature template.

4.1.3 Unsafe scenarios. The first release of Reminder-PL aimed to implement the base version of the SPL, composed of seven mandatory features that generate a single product without any variability. However, even with the inclusion of new mandatory features during the implementation of the release, it is not possible to guarantee, as defined in the Add New Mandatory Feature template, that the evolution has been safely carried out. This is because the insertion of any of the mandatory features would change the behavior of the only existing product.

The introduction of the Date Repeat feature, in the last release of Reminder-PL, does not constitute a safe evolution scenario also, due to a refactoring in the delta modules to extract common code snippets among the Reminder Schedule features. This way, two deltas that make up the Date Repeat implementation are present in other features, such as Fixed Date and Date Repeat. The Add New Alternative Feature template states that, for the evolution to be considered safe, it is necessary to ensure that the new assets are only present in products that contain the new added feature, which is not the case in this scenario.

In Iris-PL, several transactions (groups of related commits) could not be mapped to existing safe or partially safe evolution templates. For instance, the first release had 18 transactions, and only 13 of

them had been mapped into either safe or partially safe evolution scenarios. In the second and third releases, of the 7 groups of commits, 4 of them could not be mapped to any existing template.

4.1.4 Summary of the Findings. Our results show that the existing safe and partially safe evolution templates in the literature, which have been used to justify the evolution of annotative and compositional SPLs, can also be used to characterize the evolution of delta-oriented SPLs. Therefore, here we generalize the usage of existing templates for safe and partially safe evolution for transformational techniques [23, 33, 38]. Table 4 presents the templates identified in Reminder-PL and Iris-PL, respectively, as well as the number of occurrences in every analysed *release*.

SPL	TEMPLATE	SCENARIO	TOTAL
Reminder-PL	Add New Optional Feature	Safe	4
	Add New Alternative Feature	Safe	2
	Remove Unused Assets	Safe	1
	Refine Asset	Safe	1
	Change Asset	Partially Safe	4
Reminder-PL	Add New Optional Feature	Safe	4
	Add New Alternative Feature	Safe	2
	Refine Asset	Safe	4
	Change Order	Safe	1
	Feature Renaming	Safe	1
	Asset Name Renaming	Safe	1
	Add New OR Feature	Safe	1
	Change Asset	Partially Safe	7

Table 4: Occurrences of templates identified in Reminder-PL and Iris-PL.

Some of the results of other studies that have investigated and characterized safe and partially save evolution scenarios show similarities with the results presented here. For instance, templates such as Add New Optional Feature and Add New Alternative Feature, which were the most frequent in both of the analysed SPLs, were also identified in five other analysed SPLs in a study which aimed to identify and analyze concrete safe evolution scenarios [28]. In the same study, an analysis showed that the Refine Asset template had the most occurrences between the templates, contrasting with the number of occurrences in this study, which was inferior to the templates for adding features.

Regarding partially safe evolution, two other works [18, 32] showed that the Change Asset template had the higher number of occurrences, which was also observed in our results. This result is not a surprise, since it is unlikely that an evolution scenario consisting of only changing assets will not occur during the SPL's evolution process, since asset changes are essential for maintaining the product line. Indeed, we notice a higher prevalence of safe evolution scenarios, probably because our study mainly consists of evolution scenarios that would be associated with establishing an SPL from scratch. Therefore, we observe a higher number of new features being added, in contrast with an established and mature SPL, in which there might be more bug fixes and changes to existing features, which would be associated with partially safe evolution.

4.2 Answers to Research Question (RQ2): Idioms for modularizing feature interactions

To understand the interactions between features of both SPLs, we compute the occurrence of *Type (a)* and *Type (b) feature interactions*, based on the definitions presented in Section 3.3. Subsequently, we analyze these interactions in order to identify code idioms we explored to cope with the feature interaction scenarios.

In general, we identified *Type (a) feature interactions* in all releases of both SPLs. During the analysis, we realized that a single delta module can contain one or more interactions—overall, the average interactions per delta module was 3.0 for Iris-PL and 6.72 for Reminder-PL. Figure 6 summarizes the number of *Type (a) feature interactions* we found in both SPLs.

We also identified *Type (b) feature interactions* in almost all releases of both SPLs. In general, four interactions were found for each release of both SPLs, except for the last release (v4), which had a total of 5 interactions. It is important to note that this type of feature interaction requires additional lines of code, since each delta module contains common source code of the two features that interact with each other— and it is hard to factor out the common code to remove this kind of code duplication.

Figure 7 presents an example of *Type (b) feature interaction*. In the example, an instance of the Reminder-PL with the Google Calendar feature must include a method to add a reminder to the calendar. The code in blue must be included if the instance is also configured either with the Fixed Date feature or Date Range feature; while the code in green must be included only if the instance is configured with the Date Range feature. In this way, the code snippets referring to the Reminder Schedule feature are inserted in specific places of the method. As a consequence, it is necessary to create a new delta module to support the interaction between Reminder Schedule and Google Calendar features.

We also analyzed the source code of each interaction, in order to identify common code idioms we used to modularize feature interactions. As such, we identified three recurring patterns.

- **P1 Pattern** — A modification to a given class is performed so that the body of a method is modified to include an `original()` statement and a new code block (*body'*).

```
adds {
    public class C {
        void method(){
            body
        }
    }
}
```

Listing 4: P1 - Before

```
modifies C {
    modifies method() {
        original();
        body'
    }
}
```

Listing 5: P1 - After

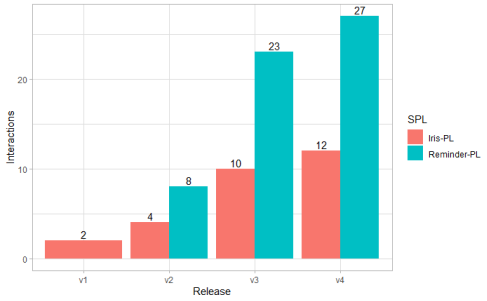

Figure 6: Summary of the *Type (a) feature interactions* in both SPLs.

Legend: GOOGLE CALENDAR | FIXED DATE or DATE RANGE | DATE RANGE

```
1  public class CalendarEventCreator {
2      public void addEvent(Reminder reminder){
3          Calendar start;
4          Calendar end;
5
6          boolean haveMainCalendar = Controller.calendar();
7          if (haveMainCalendar){
8              Date dateStart = reminder.getDateStart();
9              start.setTime(dateStart);
10
11             Date dateEnd = reminder.getDateStart();
12             end.setTime(dateEnd);
13
14             ContentValues values = new ContentValues();
15             values.put(Events.DTSTART, start);
16             values.put(Events.DTEND, end);
17             values.put(Events.TITLE, reminder.getText());
18
19             ContentResolver cr = ctx.getContentResolver();
20             cr.insert(Events.CONTENT_URI, values);
21         }
22     }
23 }
```

Figure 7: Example that motivates the creation of new delta modules to support the interaction of type (b).

- **P2 Pattern** — In this pattern, the structure of a class is modified, so that the body of a method is completely changed to include a new block of code (*body'*).

```
adds {
    public class C {
        void method() {
            body
        }
    }
}
```

Listing 6: P2 - Before

```
modifies C {
    modifies method() {
        body'
    }
}
```

Listing 7: P2 - After

- **P3 Pattern** — Represents adding a *class body declaration* (either a method or field) to a given class. When this occurs, developers often include a new entry into the `import` list of a module.

```
adds {
    public class C {
      fields
      methods
    }
}
```

Listing 8: P3 - Before

```
modifies C {
    // adds new methods or fields
    adds ClassBodyDec;
}
```

Listing 9: P3 - After

With these idioms, we better understand the structure of interactions between features, since an interaction can occur at different structural levels of a class in a delta module—from modifying a method to adding a class body declaration. Figure 8 summarizes the number of occurrences of each pattern. It is important to note that each feature interaction might require the use of more than one pattern.

To sum up, we found 39 feature interactions in both product lines (Reminder-PL and Iris-PL), in their last release. We could trace our implementations to deal with every feature interaction to one of these three code idioms (patterns).

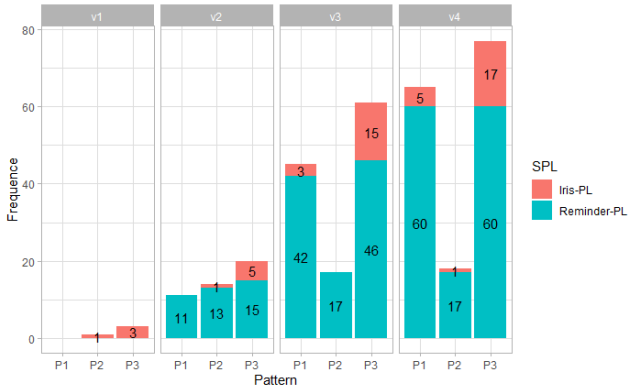

Figure 8: Patterns frequency in Reminder-PL and Iris-PL.

5 RELATED WORK

In this section, we discuss related work, in the context of delta-oriented programming, safe and partially safe evolution, and feature interactions.

Delta-Oriented Programming. Since the initial papers introducing [33], studies have emerged with the purpose of investigating the benefits of using delta-oriented constructs in SPL engineering. Here we discuss a few works that focus on product line evolution. The work of Diniz et al. [17] focused on assessing the stability of delta-oriented SPLs through evolutionary scenarios based on two implementation strategies: one that starts from a simple core implementation and another that starts from a more complete implementation of a core module (named complex core). According to the authors, the simple core approach brings more benefits in terms of stability when evolving delta-oriented SPLs. Another work, by Hamza et al. [20], conducted a case study on the unanticipated evolution of a SPL using two approaches: separate products versus a common codebase approach based on a delta-oriented SPL. The authors report that both approaches present strengths and weaknesses, and, according to the authors, DOP is not mature enough to handle complex SPLs. In our work we reach a different conclusion, that is, DeltaJ 1.5 does allow us to manage variability in SPLs an order of magnitude more complex than the vending machine used in their work [20]. Differently from this previous study, we were not able to avoid all code duplication in DOP, particularly when considering feature interactions. In addition to empirical assessments, the delta-oriented community has proposed solutions for testing DOP SPLs—as is the case with the work of Varshosaz et al.[37] and Lachmann [25] that focused on introducing new testing technologies for SPLs and the interaction between DOP and test case prioritization. The work of Damiani and Lienhardt [16] presented algorithms for refactoring delta-oriented SPLs to obtain monotonicity. Instead, here we report how one can benefit from using existing catalogs for *safe* and *partially safe* evolution of SPLs that use delta-oriented constructs.

Safe evolution. Safe evolution is a notion based on the refinement theory [12], which formalizes concepts informally presented in previous works on SPL refactoring [3, 11]. As mentioned, safe evolution is associated with evolution scenarios where the behavior of all existing products is preserved. Based on this notion, some works have investigated SPL evolution in this context, resulting in one of the main applications of such notion and underlying theory: providing support to developers in the form of templates that abstract common and recurrent safe evolution scenarios [10, 28].

The refinement theory was formalized using proof assistants [2, 12, 35]. Teixeira et al. proposed a product line of theories to reason about safe evolution [35]. Moreover, the work exploits and studies similarities among the concrete languages that specify PL elements, enabling to specify refinement templates at a higher abstraction level using the PVS theorem prover. Alves et al. present a case study on porting the PVS specification of the refinement theory to Coq [2]. This study compares the proof assistants based on the noted differences between the specifications and proofs of this theory, providing some reflections on the tactics and strategies used to compose the proof.

Partially safe evolution. Despite the fact that several changes are covered by the safe evolution notion, some common evolution scenarios remain uncovered, such as bug fixing or feature removal. Thus, previous work has extended the refinement theory with the proposal of the partially safe evolution notion [32], formalized by the partial refinement definition. This notion, as we previously mentioned, supports SPL changes preserving the behavior of only a subset of products in the SPL. In this work, Sampaio et al. also formalize partially safe templates from common scenarios (e.g., Change Asset) and provide compositionality properties for AM and CK [32]. Other works take into consideration either safe and partially safe evolution [18, 32]. Gomes et al. investigate around 2,000 commits from a SPL repository aiming to characterize the frequency and occurrences of safe and partially safe evolution scenarios. This work also categorize commits not covered by templates using characteristics of each SPL element (AM, FM, CK).

Feature Interactions. The feature interaction problem, both in general and in the particular domain of SPLs, has been extensively explored in the literature [4, 5, 13, 19, 21, 34]. For instance, Keck and Kuehn present a survey on feature interactions in telecommunication systems [21] and Robert J. Hall details 27 feature interaction scenarios in electronic mail systems [19].

Indeed, this seminal work by Robert J. Hall has inspired us in the design of the Iris-PL, with the aim of identifying feature interaction patterns using electronic mail systems. Apel et al. also use the work of Robert J. Hall to explore a technique for detecting feature interactions in SPLs [8]. Similarly, Apel et al. [7] proposed a new design paradigm in which it was possible to identify feature interactions using Alloy. In point of fact, different works on the literature focus on designing new approaches for detecting feature interactions. Soares et al. present a mapping study on this subject [34].

However, our goal here was not to identify *unknown* feature interactions, but actually to understand how we could benefit from delta-oriented constructs to modularize anticipated feature interactions. Kim et al. [22] explored the same issue, investigating the consequences of feature interactions in the context of annotation-based approaches.

6 THREATS TO VALIDITY

Similar to other empirical studies, there are a couple of threats that might limit the generalization of our study. As such, a possible threat we envision relates to the implementation strategy of the DOP SPLs, where a feature is subdivided into several modules. We believe that implementing the SPLs using another strategy could produce different results from those presented, perhaps resulting in new templates for safe and partially safe evolution and new feature interaction patterns. Another potential problem is the reliability of the manual process of identifying the feature interaction patterns. In this specific case, it may be that, for example, we have ignored some code snippets during the process of identifying the standards, and, in this way, the analysis could present different results from those we presented. To mitigate this problem, we carefully reviewed all delta modules in order to identify possible interactions.

In addition, the two SPLs we use in our investigation might not be sufficiently representative for generalizing our results. Therefore, the evolution scenarios for these SPLs might not reflect all possible existing scenarios. Although our case studies are not toying examples, we agree that Reminder-PL and Iris-PL could not be compared to other configurable systems already explored by the research community (e.g., ArgoUML and Linux as a more extreme case). Indeed, there are not many delta-oriented SPLs freely available that we could use in a mining software repository effort. Our decision here was to experiment with bootstrapping DOP SPLs from existing products and then evolving these product lines. It would be difficult to conduct this research in more complex product lines. Finally, another possible limitation of our study is the choice for DeltaJ. If we have considered other languages or tools, the findings we presented in our paper could be different.

7 FINAL REMARKS

In this work, we implemented, evolved, and analyzed two delta-oriented SPLs (Reminder-PL and Iris-PL) to better understand the implications of using DOP for software product line engineering. For that, our analysis covered different evolution scenarios (such as the inclusion of mandatory, optional, and alternative features) and the feature interactions that occur in these scenarios. The results showed that the safe and partially safe evolution templates [12, 28, 32] can be used by developers to guide the evolution of delta-oriented SPLs—in addition to proposing two new templates. We also presented three source code idioms we used to modularize several feature interactions that aroused during our experience in evolving SPLs developed in DOP. We hope that such a comprehensive experience and case studies could help practitioners and researchers to better understand the implications of using DOP to evolve SPLs.

ACKNOWLEDGMENTS

We want to thank the anonymous reviewers for their insightful comments on previous versions of this paper and Fausto Carvalho Marques Silva, Pedro Henrique Teixeira Costa, Alexandre Lucchesi, and Marcos César de Oliveira for their contributions to the implementation of Iris-PL.

REFERENCES

[1] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *2nd IEEE/ACM International Workshop on Variability and Complexity in Software Design, VACE@ICSE 2017, Buenos Aires, Argentina, May 27, 2017*. IEEE, USA, 34–40. https://doi.org/10.1109/VACE.2017.8

[2] Thayonara Alves, Leopoldo Teixeira, Vander Alves, and Thiago Castro. 2020. Porting the Software Product Line Refinement Theory to the Coq Proof Assistant. In *Formal Methods: Foundations and Applications*, Gustavo Carvalho and Volker Stolz (Eds.). Springer International Publishing, Cham, 192–209.

[3] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. 2006. Refactoring Product Lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*. ACM, New York, NY, USA, 201–210. https://doi.org/10.1145/1173706.1173737

[4] Daniel Amyot, Leïla Charfi, Nicolas Gorse, Tom Gray, Luigi Logrippo, Jacques Sincennes, Bernard Stepien, and Tom Ware. 2000. Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS.. In *Feature Interactions in Telecommunications and Software Systems*. IOS Press, Ottawa, CA, 274–289.

[5] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD '13)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2528265.2528267

[6] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. 2010. Detecting Dependences and Interactions in Feature-Oriented Design. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. IEEE Computer Society, USA, 161–170. https://doi.org/10.1109/ISSRE.2010.11

[7] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. 2010. Detecting Dependences and Interactions in Feature-Oriented Design. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. IEEE Computer Society, USA, 161–170. https://doi.org/10.1109/ISSRE.2010.11

[8] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, USA, 372–375.

[9] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, J. Henk Obbink and Klaus Pohl (Eds.). Lecture Notes in Computer Science, Vol. 3714. Springer, Switzerland, 7–20. https://doi.org/10.1007/11554844_3

[10] Fernando Benbassat, Paulo Borba, and Leopoldo Teixeira. 2016. Safe Evolution of Software Product Lines: Feature Extraction Scenarios. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2016, Maringá, Brazil, September 19-20, 2016*. IEEE Computer Society, USA, 11–20.

[11] Paulo Borba. 2011. An Introduction to Software Product Line Refactoring. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III (GTTSE'09)*. Springer-Verlag, Berlin, Heidelberg, 1–26. http://dl.acm.org/citation.cfm?id=1949925.1949927

[12] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A theory of software product line refinement. *Theoretical Computer Science* 455 (oct 2012), 2–30. https://doi.org/10.1016/j.tcs.2012.01.031

[13] E.J. Cameron and H. Velthuijsen. 1993. Feature interactions in telecommunications systems. *IEEE Communications Magazine* 31, 8 (1993), 18–23. https://doi.org/10.1109/35.229532

[14] P. Clements and L. Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley, USA.

[15] Ferruccio Damiani and Michael Lienhardt. 2016. On Type Checking Delta-Oriented Product Lines. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, Erika Ábrahám and Marieke Huisman (Eds.). Lecture Notes in Computer Science, Vol. 9681. Springer, Switzerland, 47–62. https://doi.org/10.1007/978-3-319-33693-0_4

[16] Ferruccio Damiani and Michael Lienhardt. 2016. Refactoring Delta-Oriented Product Lines to achieve Monotonicity. In *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FM-SPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016 (EPTCS)*, Julia Rubin and Thomas Thüm (Eds.), Vol. 206. 2–16. https://doi.org/10.4204/EPTCS.206.2

[17] João P. Diniz, Gustavo Vale, Felipe Nunes Gaia, and Eduardo Figueiredo. 2017. Evaluating Delta-Oriented Programming for Evolving Software Product Lines. In *2nd IEEE/ACM International Workshop on Variability and Complexity in Software Design, VACE@ICSE 2017, Buenos Aires, Argentina, May 27, 2017*. IEEE, 27–33. https://doi.org/10.1109/VACE.2017.7

[18] Karine Gomes, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro, and Rohit Gheyi. 2019. Characterizing safe and partially safe evolution scenarios in product lines. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems - VAMOS '19*. ACM Press.

[19] Robert J Hall. 2005. Fundamental nonmodularity in electronic mail. *Automated Software Engineering* 12, 1 (2005), 41–79.

[20] Mostafa Hamza, Robert J. Walker, and Maged Elaasar. 2017. Unanticipated Evolution in Software Product Lines versus Independent Products: A Case Study. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/3109729.3109739

[21] D.O. Keck and P.J. Kuehn. 1998. The feature and service interaction problem in telecommunications systems: a survey. *IEEE Transactions on Software Engineering* 24, 10 (1998), 779–796. https://doi.org/10.1109/32.729680

[22] Chang Hwan Peter Kim, Christian Kästner, and Don S. Batory. 2008. On the modularity of feature interactions. In *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, Yannis Smaragdakis and Jeremy G. Siek (Eds.). ACM, 23–34. https://doi.org/10.1145/1449913.1449919

[23] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. 2014. DeltaJ 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform Virtual machines, Languages, and Tools - PPPJ '14*. ACM Press. https://doi.org/10.1145/2647508.2647512

[24] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. 2015. Delta-oriented test case prioritization for integration testing of software product lines. In *Proceedings of the 19th International Conference on Software Product Line - SPLC '15*. ACM Press. https://doi.org/10.1145/2791060.2791073

[25] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. 2015. Delta-Oriented Test Case Prioritization for Integration Testing of Software Product Lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. Association for Computing Machinery, New York, NY, USA, 81–90. https://doi.org/10.1145/2791060.2791073

[26] Sascha Lity, Matthias Kowal, and Ina Schaefer. 2016. Higher-order delta modeling for software product line evolution. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development - FOSD 2016*. ACM Press. https://doi.org/10.1145/3001867.3001872

[27] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–150.

[28] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. 2015. Safe evolution templates for software product lines. *Journal of Systems and Software* 106 (aug 2015), 42–58.

[29] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts. *Empirical Software Engineering* 21, 4 (01 Aug 2016), 1744–1793. https://doi.org/10.1007/s10664-015-9364-x

[30] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[31] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*. Springer, 419–443.

[32] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially safe evolution of software product lines. *Journal of Systems and Software* 155 (2019), 17 – 42. https://doi.org/10.1016/j.jss.2019.04.051

[33] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented Programming of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 77–91.

[34] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology* 98 (2018), 44–58.

[35] Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. 2015. A Product Line of Theories for Reasoning about Safe Evolution of Product Lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. Association for Computing Machinery, New York, NY, USA, 161–170.

[36] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages.

[37] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. 2015. Delta-Oriented FSM-Based Testing. In *Formal Methods and Software Engineering*, Michael Butler, Sylvain Conchon, and Fatiha Zaïdi (Eds.). Springer International Publishing, Cham, 366–381.

[38] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, and Ina Schaefer. 2016. Parametric DeltaJ 1.5: Propagating Feature Attributes into Implementation Artifacts. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016 (CEUR Workshop Proceedings)*, Vol. 1559. CEUR-WS.org, 40–54.