Journal of Universal Computer Science, vol. 20, no. 5 (2014), 587-614
submitted: 28/7/13, accepted: 15/2/14, appeared: 1/5/14

A Toolset for Checking SPL Refinements

Felype Ferreira
(Federal University of Pernambuco, Recife, Brazil
fsf2@cin.ufpe.br)

Rohit Gheyi
(Federal University of Campina Grande, Campina Grande, Brazil
rohit@dsc.ufcg.edu.br)

Paulo Borba
(Federal University of Pernambuco, Recife, Brazil
phmb@cin.ufpe.br)

Gustavo Soares
(Federal University of Campina Grande, Campina Grande, Brazil
gsoares@dsc.ufcg.edu.br)

Abstract: Developers evolve software product lines (SPLs) manually or using typical program refactoring tools. However, when evolving an SPL to introduce new features or to improve its design, it is important to make sure that the behavior of existing products is not affected. Typical program refactorings cannot guarantee that because the SPL context goes beyond code and other kinds of core assets, and involves additional artifacts such as feature models and configuration knowledge. Besides that, we typically have to deal with a set of alternative assets that do not constitute a well-formed program in an SPL. As a result, manual changes and existing program refactoring tools may introduce behavioral changes or invalidate existing product configurations. To reduce such risks, we propose approaches and implement four tools for making product line evolution safer. These tools check if SPL transformations preserve the behavior of the original SPL products. They implement different and practical approximations of refinement notions from a theory for safely evolving SPLs. Besides specifying the algorithms of each approach, we compare them with respect to soundness, performance and code coverage in 35 evolution scenarios of an SPL with 32 KLOC.
Key Words: software product lines, safe evolution, refactoring, checking tools.
Category: D.2.4, D.2.5, D.2.7, D.2.13.

1 Introduction

A software product line (SPL) is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [Pohl et al., 2005, van der Linden et al., 2007]. However, SPL evolution can be quite challenging. First, changes to a given asset might affect the behavior of a number of products. Second, we have to deal not only with assets but

also with artifacts, such as feature models (FM) [Kang et al., 1990] and configuration knowledge (CK) [Czarnecki and Eisenecker, 2000], that enable product generation, and they should all be changed consistently.

During SPL evolution, it might be important to make sure that the associated changes do not affect the behavior of the existing SPL products. This may be useful when refactoring an SPL, that is, simply improving the design of its artifacts. We also need that when extending an SPL by making it able to generate new products, including new optional features, for example. We use the same notion of behavior preservation of Opdyke [Opdyke, 1992] when comparing products: for the same set of input values, the resulting set of output values should be the same. This notion of evolving an SPL without changing the behavior of existing products is captured by a formal notion of SPL refinement [Borba et al., 2012], which guarantees that the observable behavior of products in the original SPL is preserved by corresponding products in the new, evolved, SPL. However, it is time consuming and error prone to evaluate whether each SPL evolution is safe[1] with respect to the definition [Teixeira et al., 2013]. An associated catalogue of safe SPL evolution transformations [Neves et al., 2011] was proposed to reduce this problem. It can be used by developers, in the same way that object-oriented single program refactoring catalogues are available in current development environments. However, there are a number of useful transformations that cannot be justified by this catalogue. Moreover, catalogue driven evolution may not be appealing in some practical contexts.

Developers often evolve SPLs without tool support for checking that the associated changes are safe. At most, when refactoring, they rely on the support provided by typical single program refactoring tools that check a number of preconditions for behavior preservation of the modified assets. However, they not only may perform incorrect refactorings for single programs [Soares et al., 2013], but also they are unaware that SPLs often have conflicting assets that implement alternative features, and therefore do not constitute a valid program. They are also unaware of other artifacts such as FM and CK, which should be consistently changed together with the reusable assets. As a result, some transformations may change the behavior of existing products and negatively impact users.

To help developers on safely evolving SPLs, we proposed four tools for checking SPL refinement [Ferreira et al., 2012]: All Product Pairs, All Products, Impacted Products, and Impacted Classes. They implement different and practical approximations of a theory for safely evolving SPLs. The suitability of each tool depends on the kind of change an SPL is subject to, and on user's constraints regarding reliability and time.

Our previous work [Ferreira et al., 2012] evaluates the proposed tools in 15

[1] The term safe evolution that we use here is not related to system safety properties and it refers only to the behavior preservation in SPL evolution scenarios.

evolution scenarios. We extend our previous work by specifying the algorithms of each approach, and evaluating[2] our tools in 35 evolution scenarios of an SPL with 32 KLOC. We compare them with respect to soundness, performance and code coverage. This new analysis reinforces results obtained in our previous analysis and go beyond that by estimating how often the tool can lead to false positives and negatives, and better understanding in what situations they may happen.

The remaining of this article is organized as follows. Section 2 presents some problems that may happen while evolving SPLs. Section 3 describes our tools. Next, we evaluate them in Section 4. Finally, we present related work and final remarks in Sections 5 and 6, respectively.

2 Motivating Examples

When evolving an SPL, developers often manually change the different SPL artifacts like FMs [Kang et al., 1990] and reusable assets. To change the code assets, they might also use code refactoring tools. Unfortunately, this can lead to problems like the generation of ill-formed products (Section 2.1) or undesirable changes to the behavior of the existing ones (Section 2.2).

2.1 Invalid Products

Consider a simple SPL evolution scenario with a toy example of game SPL (see Figure 1). On the left hand side, an SPL contains a FM, where `Multiplayer`, `Internet` and `Bluetooth` are optional, `Startup` and `Connection` are mandatory. These five features and their relationships allow five product configurations (valid feature selections). This SPL also contains a CK, which relates feature expressions to sets of asset names, linking solution and problem spaces. For example, the first row relates the joint selection of features `Game` and `Startup` to the `Game.java` and `Startup.java` names. To generate a product, we evaluate the CK against a valid, accordingly to the FM, product configuration. Evaluating this CK with the product configuration {`Game`, `Multiplayer`, `Startup`, `Bluetooth`} yields the following set of asset names {`Game.java`, `Multiplayer.java`, `Startup.java`, `Bluetooth.java`}. Besides FM and CK, this SPL declares code assets, as shown in Figure 1. Notice that some assets depend on other ones. For instance, the `Bluetooth` class extends the `Multiplayer` class. The CK must be correctly specified to avoid sets of asset names that do not represent well-formed products due, for example, to missing dependences.

Suppose we apply the *Pull up field* refactoring to move the field `Internet.con` to the `Multiplayer` class using the Eclipse refactoring implementation. It performs the change without any warning. However, this evolution

[2] All experiment data are available at: `http://www.cin.ufpe.br/~fsf2/jucs_experiments.html`

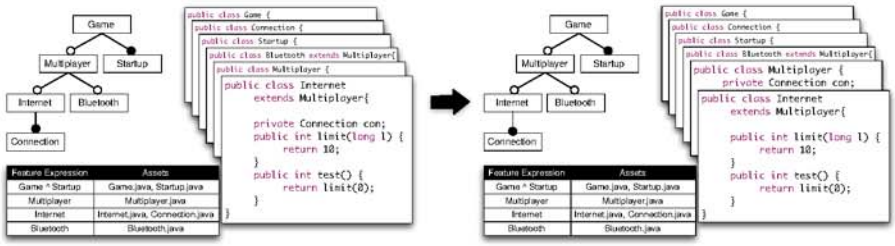

Figure 1: An SPL evolution yields invalid products.

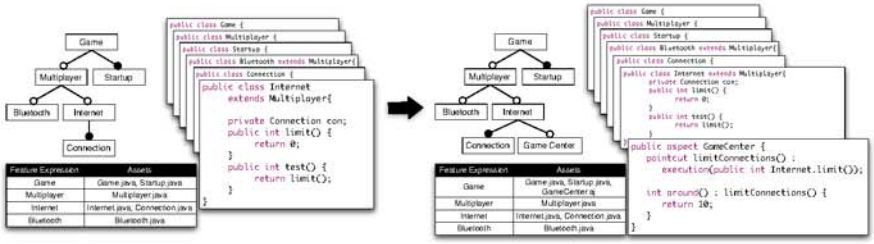

Figure 2: An SPL evolution yields behavioral changes in existing products.

step is unsafe. The resulting SPL generates invalid products, sets of assets that do not compile. The CK evaluation against the product configuration {`Game`, `Multiplayer`, `Startup`, `Bluetooth`} does not yield the `Connection` class, which the `Multiplayer` class needs to compile. In fact, an SPL aware refactoring tool would not only move the field to the superclass but also update the CK by moving `Connection.java` to the assets provided by the feature expression `Multiplayer`.

2.2 Behavioral Changes

Besides resulting in invalid products, that do not compile, changes to assets can also introduce behavioral changes to existing products. To illustrate that, consider the left hand side SPL of Figure 1. Suppose we manually add a new feature `Game Center` to the FM. To implement the new functionality we also create an aspect that changes the behavior of the method `Connection.limit()` only in the products that contain the new feature `Game Center`. However, when evolving the CK, developers might make an incorrect association. Instead of associating the new asset `GameCenter.aj` with the new feature `Game Center`, they might associate it to the root feature `Game`. Figure 2 depicts this transformation.

All products of the resulting SPL are well-formed in this transformation. However, the SPL evolution does not preserve the behavior of the existing products. Consider the `Internet` class and the `GameCenter` aspect (see Figure 2) and the product configuration {`Game`, `Multiplayer`, `Startup`, `Internet`,

`Connection`}. Whereas before the evolution, the method `Internet.test()` calls its method `limit()`, and yields 0, after that, the call to is affected by the around implemented in the new aspect, and yields 10. Therefore, after the evolution scenario, all products contain an asset of the optional feature `Game Center` and present different behavior when executing the `test` method.

3 Tool Support for Checking SPL Refinements

In this section, we describe our toolset for checking if a transformation applied to an SPL is safe. They implement different and practical approximations of the theory for safely evolving SPLs. The suitability of each tool depends on the kind of change an SPL is subject to. For instance, if we change only the FM and CK, we can apply faster tools. Also, although slower, some tools are more reliable than others. We propose four tools, named after the approaches they implement:

- All Product Pairs checks, for each original product, if after the transformation there is a product with compatible observable behavior (Section 3.1);
- All Products is similar to the All Product Pairs, however it only compares original products with resulting products with the same set of asset names, simplifying the checking (Section 3.2);
- Impacted Products checks only the products impacted by the change. It potentially analyzes fewer products than the All Product Pairs and All Products tools, reducing the time to check the refinement (Section 3.3.1);
- Impacted Classes focuses on testing only the changed assets. By doing so, it avoids generating and testing all impacted products, which can lead to a major reduction on time compared to the first three tools (Section 3.3.2).

3.1 All Product Pairs (APP)

All Product Pairs is our baseline tool. It directly checks the SPL refinement definition, looking for corresponding products with compatible observable behavior, which are products that behaviorally match when we compare them. Next, we explain its process, described in Algorithm 1.

The *Step 1* of the tool checks if the SPL after the transformation (*target SPL*) is well-formed (Line 2), which means that it still generates well-formed products, that correspond to valid products in the underlying languages used to describe assets. If it finds a problem, it stops the process (Line 2), and indicates that the SPL is not refined and the evolution is unsafe (Line 21). Additionally, the tool can report all the invalid product configurations found. If it does not find a problem, in *Step 2*, it checks whether, for each product in the original

SPL (*source SPL*), there is a product in the resulting SPL (*target SPL*) with equivalent behavior (Lines 3-19). The *Step 2.1* analyzes for a product *ps* in the source SPL, if there is a product *likely* with the same set of asset names in the target SPL and maps it as the *likely corresponding target product* (Line 4). After mapping products, in *Step 2.2*, it checks whether the product in the source SPL and its likely corresponding, when it exists, have compatible observable behavior (COB) using randomly generated unit test cases (Line 5). Exceptionally, when their observable behavior are not compatible or the source configuration does not exist in the target SPL, the tool performs *Step 2.3*: it compares the behavior of the source product against all the other target products (Lines 7-15). If it does not find any corresponding product, it assumes that the SPL is not refined (Line 16), reporting the first occurrence of product configuration without corresponding refined product, and the set of tests that reveals the behavioral changes in this product. Otherwise, when it finds corresponding products for all source products, we can increase our confidence that the evolution is an SPL refinement (safe evolution).

Algorithm 1: The ALL PRODUCT PAIRS checking process.

```
   Input: source <-- Source SPL, target <-- Target SPL
   Output: True if target refines source, False otherwise
 1 refinement <-- False
 2 if wf(target) then
 3     foreach product ps in source do
 4         likely <-- likely corresponding(ps)
 5         refinement <-- likely ∧ cob(ps, likely)
 6         if !refinement then
 7             foreach product pt in target do
 8                 if pt != likely then
 9                     if cob(pt, ps) then
10                         refinement <-- True
11                         break
12                     end
13                 end
14             end
15         end
16         if !refinement then
17             break
18         end
19     end
20 end
21 return result
```

To illustrate it, consider the motivating examples in Section 2. For the first example (Section 2.1), in Step 1 the ALL PRODUCT PAIRS tool reports that 2 out of 5 product configurations ({`Game`, `Multiplayer`, `Startup`} and {`Game`, `Multiplayer`, `Startup`, `Bluetooth`}) yield sets of products that do not compile after the evolution. For the second example (Section 2.2), it does not detect problems in Step 1. Then, for each of the five products in the source SPL it analyzes whether there is a product with compatible observable behavior in the target SPL in Step 2. In this case, as we incorrectly modified the CK, the target SPL does not generate exactly the same product (set of asset names) for

each of the configurations allowed by the FM of the source SPL. For example, our tool detects that the source product generated by the configuration {`Game`, `Multiplayer`, `Startup`, `Internet`, `Connection`} does not have a corresponding product in the target SPL with the same assets (Step 2.1). Therefore, it does not applies Step 2.2, and in Step 2.3, it uses unit test cases to compare this product against all other target products but does not find any behaviorally compatible with it. Therefore, the All Product Pairs tool reports that the evolution scenario is unsafe. Next we give more details about the implementation.

3.1.1 Implementation

The function wf in Algorithm 1, Line 2, checks whether the SPL is well formed. We use a theory for FMs [Gheyi et al., 2006] and CKs [Teixeira et al., 2013] encoded in Alloy [Jackson, 2006], a formal specification language, to check whether an SPL is well formed. Our tools translate the FM and CK into this theory, and we use the Alloy Analyzer tool to perform analysis.

In Line 4, All Product Pairs generates product configurations and maps source products to their likely corresponding products when they exist. We use the Alloy Analyzer for generating the product configurations from the source FM. Then, we construct source and target sets of assets using the source and target FM and CK.

To check behavioral changes (the function *cob* in Line 5) we use SafeRefactor [Soares et al., 2010, Mongiovi et al., 2014], a tool for checking behavioral changes. First, the tool checks for compilation errors in the resulting program, and reports them; if no errors are found, it analyzes the transformation and generates a number of tests suited for detecting behavioral changes. SafeRefactor identifies the methods with matching signature before and after the transformation. Next, it applies Randoop [Pacheco et al., 2007], a random unit test generator for Java, to produce a test suite for these methods. Randoop randomly generates tests for classes within a time limit. A test typically consists of a sequence of method and constructor invocations that creates and mutates objects with random values, plus an assertion. Finally, it runs the tests before and after the transformation, and evaluates the results. If results are divergent, it reports a behavioral change.

To illustrate SafeRefactor, consider the behavioral change in the product configuration {`Game`, `Multiplayer`, `Startup`, `Internet`, `Connection`}, explained in Section 2.2. After the transformation, the `Internet.limit()` method yields 10, instead of 0. SafeRefactor first identifies the methods with matching signatures on both versions. In this case, it identifies all methods since there was no change in their signatures. Next, it generates unit tests for these methods by using Randoop. Finally, it runs the test suite on both versions and evaluates the results. Next, we show one of the generated tests that revealed behavioral

changes. The test passes in the source program since the value returned by `Internet.limit` is 0; however, it fails in the target program since the value returned by `Internet.limit` is 10.

Listing 1: Test case generated by Randoop that reveals a behavioral change.

```
public void test() {
  Internet var0 = new Internet();
  int x = var0.limit(0);
  assertTrue(x == 0);
}
```

We make a small modification in SafeRefactor to compare SPL products behavior with All Product Pairs. Its original version identifies the public methods in common between source and target programs and compares their behavior with respect to these methods [Soares et al., 2010]. Since we compare products with different configurations, they may have different features. By comparing them with respect to the common methods, we would not be considering the methods that implement features only present in one of the versions, which would lead to false positives: the All Product Pairs tool would report a refinement when source and target products have different behavior. It is important to avoid false positives since it may hide bugs introduced to the products.

To avoid this kind of false positive, when the tool detects that the source and target products have different public methods, it considers that they do not have compatible observable behavior, as in the case that the features present in one product actually provide behavior that is not implemented by the other product. Otherwise, the tool continues the evaluation of the products.

Although this change avoids false positives, it can generate false negatives, that is, the All Product Pairs tool reports that the source and target products do not have compatible observable behavior, but they do have, when public methods are removed or added. This situation, may happen, for example, when we remove dead code of a project, or when we add methods, but we do not immediately use them in the SPL. False negatives may slow the development process, since developers would need to manually evaluate the transformation to be sure that the transformation would preserve products' behavior.

The All Product Pairs supports feature renaming. For instance, suppose we rename a feature `F` to `G`, and update the CK, changing from `F -> C.java` to `G -> C.java`. By doing so, source and target SPLs will generate products with exactly the same set of assets. Therefore, All Product Pairs will find a likely corresponding product for each source product so that it can compare its behavior, finding no behavioral change. On the other hand, for method and class renaming, the tool may generate false negatives as explained before.

Notice that All Product Pairs checks an approximation of the SPL refinement definition [Borba et al., 2012] since tests cannot prove the absence of

behavioral changes. A full guarantee cannot often be given using tests since, in general, the equivalence and refinement of observational behavior are undecidable, and the notion of SPL refinement relies on such a notion of behavioral preservation. However, with our tools, developers can improve confidence whether a transformation is safe.

3.2 All Products (AP)

The All Products tool relies on the intuition that if a product is not refined by its likely corresponding product, it is probably not refined by any other product. We describe its process in Algorithm 2. Differently from All Product Pairs, All Products does not contain Step 2.3: when some source product and its likely corresponding target product do not have compatible observable behavior, this tool immediately assumes a non refinement and does not compare the source product with the other target products (Lines 6-7). Since the All Products can only behaviorally compare source and target products with the same sets of assets, if some source product does not have a likely corresponding product in the target line, All Products is not applied (Line 1).

Algorithm 2: The All Products checking process.

```
    Input: source ←− Source SPL, target ←− Target SPL
    Output: True if target refines source, False otherwise
 1  if For each product in source exists a likely corresponding product, with the same asset names, in target then
 2      refinement ←− False
 3      if wf(target) then
 4          foreach product ps in source do
 5              likely ←− likely corresponding(ps)
 6              refinement ←− cob(ps, likely)
 7              if !refinement then
 8                  break
 9              end
10          end
11      end
12  end
13  else
14      Interrupt the execution and report an error: It is not possible to apply this tool.
15  end
16  return result
```

Since this tool does not compare products with different configurations, it does not need to worry about different public methods, and compare source and target products even when they have different public methods, avoiding false negatives when, for instance, developers rename a method. Therefore, differently from All Product Pairs, this tool supports method renaming.

However it does not look for another product in the target SPL that may have compatible observable behavior of the source product and, because of this, may lead to false negatives too. They may occur, for example, after renaming a class: the some source product will have a corresponding product in the target

SPL but source and target products have different sets of asset names due to the rename class refactoring. Therefore, similar to All Product Pairs, this tool also supports feature renaming and does not support rename class refactoring.

3.3 Optimized Approaches

For SPLs containing hundreds of products, the previous tools may not be appropriate since they may take a long time to perform the analysis. Impacted Products and Impacted Classes contain further optimizations to reduce the cost for checking SPL refinement. They analyze the transformation and use some refinement properties [Borba et al., 2012] that simplify checking.

First, both tools introduced in this section check if the target SPL is well-formed (Step 1). Then, they suppose that source and target SPLs differ only with respect to the FMs and CKs and bypass asset and product refinement checking. Next, they evaluate the CK for every possible configuration present in FM, checking if all existing evaluations of CK with the configurations of FM are still present in the evaluations of the resulting CK and FM. In this case, the target FM and CK jointly refine the source FM and CK (Step 2) [Borba et al., 2012]. If this condition is not satisfied, we can only apply All Product Pairs to check the evolution scenario. Otherwise, the optimized tools can be applied. To do so, after this checking, both tools analyze if there are changes in the code assets (Step 3). If they do not find changes, they assume that the SPL is refined; otherwise they check the SPL refinement as described in the next sections (Step 4).

For instance, consider the FM of our motivating example (see Figure 1). Suppose we change the mandatory feature `Startup` to optional. Since this transformation just changes the FM, we can perform Steps 1 and 2 only. The tools detect that the resulting sets of assets contain the original ones. Since the code did not change, they assume that the evolution scenario is safe. In the same way, they support feature rename refactoring. However, if the transformation, besides containing a feature rename refactoring, also contains a class renaming, changing the set of assets, it is impossible to check if the FM and CK are refined, making this scenario not possible to be checked by Impacted Products and Impacted Classes. Next, we describe both tools in more details.

3.3.1 Impacted Products (IP)

Besides using FM and CK optimizations, the Impacted Products tool optimizes All Products. It does so by only evaluating products that contain the changed assets, since the remaining products continue unchanged. We describe this optimized process in Algorithm 3. Notice that Line 1checks if the target FM and CK jointly refine the source FM and CK. In Line 6, the function $isDiff$ compares the source product against its likely corresponding product to check

if there is any code change in their assets . Only if it finds such a change, it will check whether both products have compatible observable behavior (Line 7). Therefore, it only evaluates source products containing at least one changed asset. We implemented an Abstract Syntax Tree (AST) comparator to compare each version of the code assets.

Algorithm 3: The IMPACTED PRODUCTS checking process.

```
    Input: source ←− Source SPL, target ←− Target SPL
    Output: True if target refines source, False otherwise
 1  if FMCKrefinement then
 2      refinement ←− False
 3      if wf(target) then
 4          foreach product ps in source do
 5              likely ←− likely corresponding(ps)
 6              if isDiff(ps,likely) then
 7                  refinement ←− cob(ps, likely)
 8              end
 9              else
10                  refinement ←− True
11              end
12              if !refinement then
13                  break
14              end
15          end
16      end
17  end
18  else
19      Interrupt the execution and report an error: It is not possible to apply this tool.
20  end
21  return result
```

To illustrate this tool, suppose that instead of applying the *Pull up field* refactoring to our first motivating example (see Section 2), we applied a *Move method* refactoring to `Internet.limit(long)`, moving it to the `Connection` class. Only classes `Internet` and `Connection` are modified: the `limit(long)` method is moved from `Internet` to `Connection`. From Figure 1, we see modified classes are related to the products {`Game`, `Multiplayer`, `Startup`, `Internet`, `Connection`} and {`Game`, `Multiplayer`, `Startup`, `Internet`, `Connection`, `Bluetooth`}. These products are the impacted products for this evolution scenario. The IMPACTED PRODUCTS tool then, differently from ALL PRODUCT PAIRS, only evaluates products that contain at least one changed asset, reducing the number of evaluated source products from `5` to `2`. In the worst case, the IMPACTED PRODUCTS tool checks as many products as ALL PRODUCTS. This can happen, for instance, when changed assets are related to the root feature.

3.3.2 Impacted Classes (IC)

Besides using FM and CK optimizations, the IMPACTED CLASSES tool optimizes the code assets checking. In some evolution scenarios, when developers change the assets, we only need to ensure that the transformed assets refine the original ones to check the SPL refinement [Borba et al., 2012], and therefore are trivially

refined by their counterparts in the target SPL. Since we do not evaluate whole products, like Impacted Products, All Product Pairs and All Products do, but only the changed classes, Impacted Classes tends to be faster.

Algorithm 4 formalizes the process of Impacted Classes. First, the Impacted Classes checks if the target FM and CK jointly refine the source FM and CK (Line 1). If so, the tool identifies modified assets (Line 4). For each one, the tool computes its dependences, that is, the set of other assets needed to compile the modified asset (Lines 6-7). We call this set of modified assets with their dependences as a *sub product*. Our tool compiles the source and target versions of each sub product. In our implementation, we reuse the compiled classes that belong to more than one sub product. This way, we save time needed to compile those classes and further optimize the checking. It then checks, for each source sub product, whether it has compatible observable behavior with its target sub product, generating test only for changed classed (Line 10).

Algorithm 4: The Impacted Classes checking process.

```
   Input: source ←− Source SPL, target ←− Target SPL
   Output: True if target refines source, False otherwise
 1 if FMCKRefinement then
 2     refinement ←− False
 3     if wf(target) then
 4         classes ←− changed classes(source, target)
 5         foreach class c in classes do
 6             sc ←− c in source
 7             tc ←− c in target
 8             ssubproduct ←− sc + dependences(sc)
 9             tsubproduct ←− tc + dependences(tc)
10             refinement ←− cob(ssubproduct, tsubproduct)
11             if !refinement then
12                 break
13             end
14         end
15     end
16 end
17 else
18     Interrupt the execution and report an error: It is not possible to apply this tool.
19 end
20 return result
```

For instance, consider the *Move Method* refactoring used to illustrate the previous tool (see Section 3.3.1). This refactoring changes the assets without modifying the CK and the FM. The tool computes dependences for `Internet` and `Connection` classes. For instance, the `Internet` class extends the `Multiplayer` class, and has a field of type `Connection`. Using this approach, the tool computes dependences for each identified dependence. For this class, the tool generates a sub product containing the following set of classes: {`Internet.java, Multiplayer.java, Connection.java`}.

As we can see, this tool only checks the modified classes and does not generate all products impacted by the change, optimizing the evaluation. However, it is important to mention that although costly, we can use All Product Pairs

to check any kind of evolution scenarios, while Impacted Products and Impacted Classes are suitable only when FM and CK are refined [Borba, 2011]. Moreover, with Impacted Classes we may lose precision, since local changes in OO classes may indirectly impact other ones [Ren et al., 2004], and this tool just focuses only on changed classes.

Additionally, since Impacted Classes does not generate products, we give special attention to assets implemented with conditional compilation, which need pre-processing to generate valid classes. So, we look for pre-processor directives in the modified classes and their dependences. Using the FM and CK, we get all possible combinations for these directives. Finally, we pre-process source and target grouped classes for each combination, and use SafeRefactor for checking behavioral changes. We deal with Aspects in the same way we deal with conditional compilation blocks. Based on FM and CK, we get all possible combinations that can affect the sub product and use SafeRefactor for checking behavioral changes with each of them.

4 Evaluation

In our previous work [Ferreira et al., 2012], we evaluated our toolset in 15 transformations applied to two SPLs: TaRGeT [Ferreira et al., 2010], a tool that automatically generates functional tests from use case documents written in natural language; and MobileMedia [Figueiredo et al., 2008], an SPL for applications that manipulates music, video and photo on mobile devices.

In this article, to reinforce our previous findings and observe new factors, we evaluate our toolset in 35 new scenarios applied to the TaRGeT SPL gathered from its SVN repository. First, we show the experiment definition (Section 4.1) and planning (Section 4.2). Then, we present the experiment operation and its results (Section 4.3). We interpret the results and discuss them in Section 4.4, and answer our research questions in Section 4.5. Finally, we present some threats to validity of the experiment (Section 4.6).

4.1 Definition

The goal of this experiment is to analyze four approaches (All Product Pairs, All Products, Impacted Products and Impacted Classes) for the purpose of evaluation with respect to identifying safe evolution scenarios from the point of view of researchers in the context of the TaRGeT SPL repository. In this experiment, we address the following research questions:

– **Q1**. Do the approaches correctly classify the evolution scenarios?

  For each approach, we measure the true positive rate (*recall*) and the false positive rate (*precision*). $tPos$ (true positive) and $fPos$ (false positive) represent the correctly and incorrectly safe evolution scenarios, respectively. $tNeg$

(true negative) and $fNeg$ (false negative) represent correctly and incorrectly identified unsafe evolution scenarios, respectively.

Our hypothesis is that All Product Pairs, All Products, and Impacted Products should have the same precision, since these optimizations should not affect the ability of detecting behavioral changes. We believe, though, that Impacted Classes may have a lower precision, since it may miss behavioral changes (see Section 3.3.2). On the other hand, we believe that All Product Pairs may have lower recall than the other ones, since it may generate false negatives when there is no target product with the same set of assets of a source product (see Section 3.1.1).

$$recall = \frac{\#tPos}{\#tPos + \#fNeg} \quad precision = \frac{\#tPos}{\#tPos + \#fPos} \tag{1}$$

– **Q2**. Do the approaches have the same performance?

For each approach, we measure the time required to analyze each transformation. We strongly believe that our optimizations will have better performance than our naive tool (All Product Pairs).

– **Q3**. Do the approaches have the same code coverage?

For each approach, we measure the statement code coverage in the classes modified by the transformation. We believe that Impacted Classes may have better code coverage in the modified classes since it focuses on testing only these classes.

4.2 Planning

In this section, we describe the subjects used in the experiment, the experiment design, and its instrumentation.

4.2.1 Selection of Subjects

We analyzed one Java SPL: TaRGeT. We randomly selected from TaRGeT's SVN repository 35 evolution scenarios manually applied to the TaRGeT SPL during its development. Table 1 indicates the version analyzed and the artifacts changed in the transformation. For each randomly selected version, we take its previous version to analyze whether they have the same behavior. For instance, we evaluate Version 77 of TaRGeT and the previous one (76).

4.2.2 Experiment Design

In our experiment, we evaluate one factor (approaches for detecting safe evolution scenarios) with four treatments (All Product Pairs, All Products,

		Baseline	APP	AP	IP	IC
Pair of commits	Changed Art.	Safe?	Safe?	Safe?	Safe?	Safe?
01	Assets	Yes	Yes	Yes	Yes	Yes
02	FM	Yes	Yes	Yes	Yes	Yes
03	Assets	Yes	Yes	Yes	Yes	Yes
04	Assets	Yes	Yes	Yes	Yes	Yes
05	Assets	Yes	Yes	Yes	Yes	Yes
06	Assets	Yes	Yes	Yes	Yes	Yes
07	Assets	Yes	Yes	Yes	Yes	Yes
08	Assets	No	Yes	Yes	Yes	Yes
09	Assets	No	No	No	No	No
10	Assets	No	No	No	No	No
11	Assets	Yes	Yes	Yes	Yes	Yes
12	Assets	No	Yes	Yes	Yes	Yes
13	Assets	Yes	No	Yes	Yes	Yes
14	Assets	Yes	Yes	Yes	Yes	Yes
15	Assets	Yes	Yes	Yes	Yes	Yes
16	Assets	Yes	Yes	Yes	Yes	Yes
17	Assets	Yes	Yes	Yes	Yes	Yes
18	Assets	No	No	No	No	No
19	Assets	Yes	Yes	Yes	Yes	Yes
20	Assets	Yes	Yes	Yes	Yes	Yes
21	Assets	Yes	Yes	Yes	Yes	Yes
22	Assets	Yes	Yes	Yes	Yes	Yes
23	Assets	Yes	Yes	Yes	Yes	Yes
24	Assets	Yes	Yes	Yes	Yes	Yes
25	FM	Yes	Yes	Yes	Yes	Yes
26	CK	Yes	Yes	Yes	Yes	Yes
27	Assets	Yes	Yes	Yes	Yes	Yes
28	FM	Yes	Yes	Yes	Yes	Yes
29	FM, CK	Yes	Yes	Yes	Yes	Yes
30	Assets	Yes	Yes	Yes	Yes	Yes
31	FM, CK, Assets	Yes	Yes	Yes	Yes	Yes
32	Assets	Yes	Yes	Yes	Yes	Yes
33	Assets	No	No	Yes	Yes	Yes
34	Assets	No	No	No	No	No
35	FM	Yes	–	–	Yes	Yes
	Precision		0.93	0.9	0.9	0.9
	Recall		0.96	1	1	1
	Accuracy		0.91	0.91	0.91	0.91

Table 1: Evaluation of 35 randomly chosen commits of TaRGeT.

Impacted Products and Impacted Classes). We choose a paired comparison design for the experiment, that is, the subjects are applied to all treatments. Therefore, we perform the approaches under evaluation in the 35 pairs of versions. The results can be "Yes" (safe evolution scenario) and "No" (unsafe evolution scenario).

4.2.3 Instrumentation

We used a modified SafeRefactor 1.3.3 with default configuration, and setting Randoop to avoid generating non-deterministic test cases. We used Emma 2.0 to collect the block coverage of the generated tests. SafeRefactor may have different results each time it is executed due to the randomly generation of the test suite. So, we execute it up to five times in each version. If none of the executions finds a behavioral change, we classify the transformation as a safe evolution scenario. Otherwise, we classify it as unsafe evolution. We defined a maximum number of tests to generate based on the number of methods to test for each pair of products, generating two tests per method. Since each SPL can generate a number of products, it would be difficult to set a time limit to evaluate each subject. We generated the number of tests proportional to the number of methods based on previous experiences with SafeRefactor. We use the FM and CK available from TaRGeT's SVN history to generate TaRGeT products.

Since we previously do not know which versions contain safe evolution scenarios, the first author of this article manually analyzed the transformations according to the refinement theory [Borba et al., 2012] and also compared the results of all approaches in all transformations to derive a *Baseline*. The manual analysis was based on the commit notes written by the developers and the source code, FM and CK comparison (diff) between the versions using the Eclipse IDE.

4.3 Operation

We ran our experiment on a quad-processor 2.66-GHz Server with 8 GB of RAM running Ubuntu 10.04. Table 1 presents the results of our evaluation. The column *Baseline* indicates whether the pair is a safe evolution based on all results, as explained in Section 4.2.3. The following columns represent the results of each approach. Cells in dark gray show incorrect results according to the baseline. At the bottom of the table, we show the precision, recall, and accuracy of each approach with respect to the column *Baseline*.

All Product Pairs yielded two false negatives because it classified two safe evolutions as unsafe (see Table 1). All Products, Impacted Products and Impacted Classes correctly categorized 31 transformations. On the other hand, in three transformations, they yield false positives. The last pair of TaRGeT could not be analyzed by All Products and All Product Pairs. It yields out of memory.

Moreover, we also analyze the time required by each approach to evaluate the transformation. Figure 3 shows the measured time (mean of 5 executions with standard deviation between 0.2704 and 3.3894) for checking all transformations.

Finally, we measure the code statement coverage of the tests randomly generated by each approach, as depicted by Figure 4. We only collect this information

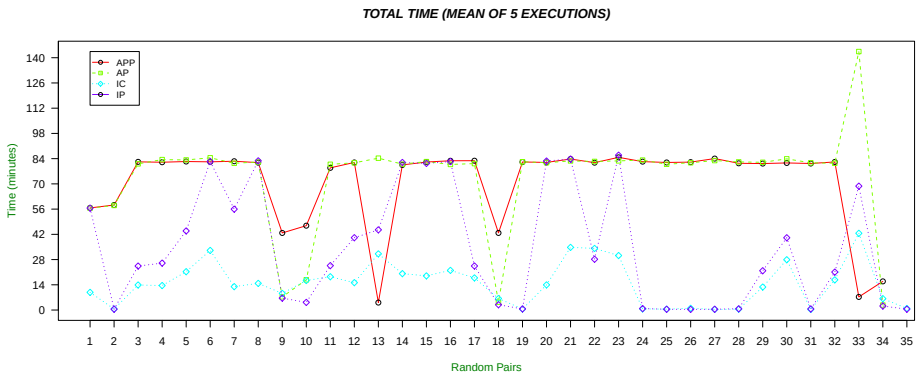

Figure 3: Time required by each tool to evaluate 35 the transformations.

in the classes modified by the transformation. We use the diff tool of Eclipse IDE to identify them. Transformations that do not change classes are not depicted by the Figure 4. All tools have almost the same results for all transformations analyzed. Pairs 13, 33 and 34 presented coverage zero for the All Product Pairs tool because it does not generate tests for these cases. Some pairs, such as Pairs 8 and 12 presented low coverage results because the changes was performed in classes that override protected methods of Eclipse RCP, like classes responsible for GUI components, whose method are called only by the framework.

4.4 Discussion

Soundness. The four tools present false positives because the changes were performed only in UI components. SafeRefactor is unable to easily detect changes in UI or output files. For instance, Pair 12 consists of a transformation performed only in UI components. In this transformation, developers would like to add more two components in the screen. Though the change is, visually, easily detected, it is not identified by Randoop because it does not affect the method outputs of the application. Besides, most of the methods that build the UI components are protected and directly called by the framework. As Randoop tests only public methods and there are no methods in the project that calls these protected methods, this kind of change cannot be identified.

Moreover, All Product Pairs presents one false negative, missing the answer just because the transformation adds public methods not called by any class. For the same transformation, All Products, Impacted Products and Impacted Classes correctly detect a behavior change. Besides, for evolutions that change the set of public methods of existing products, All Product Pairs always presents false negatives and it is not indicated for these situations. In these

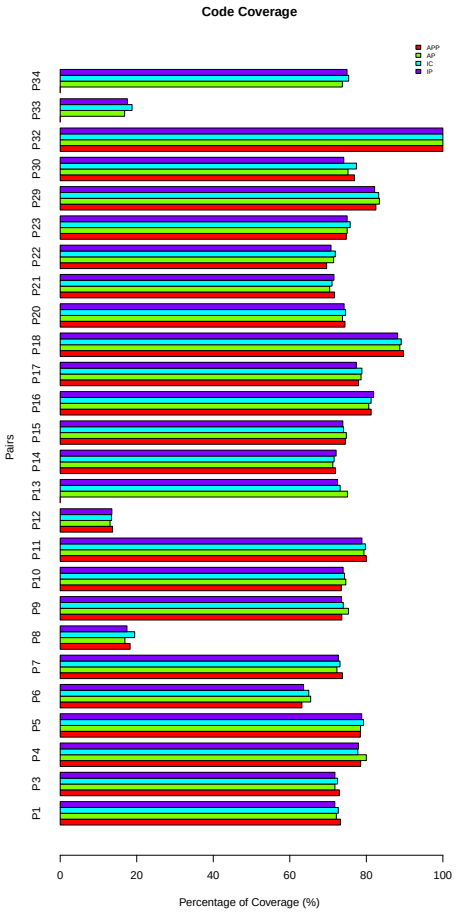

Figure 4: Code coverage in changed classes of each tool to evaluate 35 transformations applied to TaRGeT.

cases, All Products is more suitable to evaluate the transformation, though less faithful to the refinement theory.

Opdyke [Opdyke, 1992] compares the observable behavior of two programs with respect to the main method (a method in common). It checks for source and target programs that, for the same inputs, the resulting output values must be the same. SafeRefactor checks the observable behavior with respect to randomly generated sequences of methods invocations. They only contain calls to methods common to both. If the source and target programs have same results for the same input, we improve the confidence that they have equivalent behavior. Otherwise, they have different behavior. In All Product Pairs, we changed its

implementation to only check refinements between products with the same public methods, considering additions and reductions in the set of public methods a non-refinement. This implementation may be too strong in some SPLs as we saw in Pair 13. It may be better to evaluate the SPLs with respect to its Facade considering only its public methods. We could adapt SafeRefactor to evaluate the SPLs using this equivalence notion and Randoop to generate method inputs more adequate to exercise as many as possible execution flows from the Facade.

In spite of that, considering the refinements we analyzed so far, developers often add optional methods in mandatory classes using mechanisms like pre-processing and aspects to avoid including code that will only be used by optional features in all the products. Therefore, it is uncommon to include methods related to optional features in mandatory classes.

Performance. Moreover, with respect to the time required by each approach, Impacted Products and Impacted Classes are faster in analyzing Pairs 2, 19, 24 to 28, 31 and 35 because they only change FM or CK. Their original classes remain unchanged. When only some classes change, Impacted Classes tends to be faster then the other tools (Pairs 1, 3-8, 11-12, 14-17, 20-21, 23, 29-30 and 32). However, when the evolution modifies a number of classes (Pairs 9 and 10), the dependency analysis of Impacted Classes may have the same performance of Impacted Products. Additionally, in 34% of the transformations changed root features (Pairs 1, 6, 8, 9, 14, 15, 16, 18, 20, 21, 23, 34). In these pairs, Impacted Products and All Products had equivalent performance, since both tools checked the same number of products. In Pairs 13 and 33, All Product Pairs assumes a non refinement based only in changes in the set of public methods. It is faster since it does not generate a test suite.

Although we did not find performance issues of our tools due to the size of the products, we found problems related to the number of products that an SPL may generate. All Products and All Product Pairs could not analyze Pair 35, where the source SPL has more than 2,000 products to be compared. Our current implementations of these tools have limitations to analyze SPL with large number of products due to memory leak issues. As a future work, we plan to fix this issue. However, even if All Product Pairs and All Products could complete the analysis, it would take a lot of time, making clear the need for optimizations for checking product line refinements.

When the evolution scenarios are refinements, All Product Pairs and All Products tends to evaluate the same number of generated products. In these cases, their measured times are almost the same (Pairs 1-8, 11-12, 14-17 and 19-32). In Pair 33, the number of products after the transformation doubles. However, All Product Pairs took only a few minutes to conclude the checking. It happens because the transformation changed the number of public methods and the All Product Pairs tool wrongly concludes the result

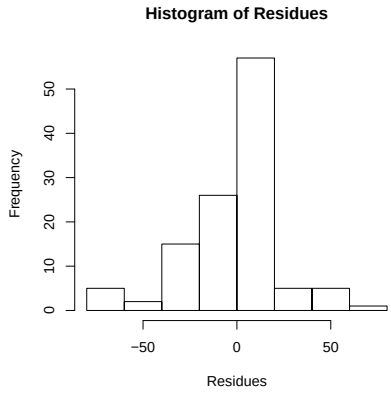

Figure 5: Histogram of residues in measured times checking pair of commits randomly chosen.

without generating tests.

We analyzed our time results using the One Way Anova Test to compare if there is significant difference between our approaches on the evaluated scenarios. Table 2 presents the results for Anova Test.

Analysis of Variance Table Response: time				
	Df	Sum Sq	Mean Sq	F value Pr(>F)
Tool	3	82099	27366.3	44.268 < 2.2e-16 ***
Residuals	132	81602	618.2	
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1)				

Table 2: Summary of One Way Anova Test results.

We used the Tukey-test to compare the means and find which tools have more significant difference in time. Table 3 presents the result for Tukey-test. Values near to zero indicate more difference between approaches. The results confirm our expectations, presenting a significant difference when comparing All Product Pairs and All Products with the optimized approaches Impacted Products and Impacted Classes. The comparison between All Product Pairs and All Products results in a low difference.

To increase the significance of the Anova table, we analyzed the normality of the residues in the measurements. Figure 5 presents a histogram of them. We can observe that they are approximately normal.

Coverage. Statement coverage is a common metric to evaluate the quality

Tukey multiple comparisons of means 95% family-wise confidence level Fit: aov(formula ∼ time ∼ tool)				
Approaches	diff	lwr	upr	p adj
APP-AP	-2.931653	-18.622849	12.75954	0.9620571
IC-AP	-58.522159	-74.213355	-42.83096	0.0000000
IP-AP	-38.376612	-54.067808	-22.68542	0.0000000
IC-APP	-55.590506	-71.281702	-39.89931	0.0000000
IP-APP	-35.444959	-51.136155	-19.75376	0.0000002
IP-IC	20.145547	4.454351	35.83674	0.0059190

Table 3: Summary of Tukey test results.

of a test suite. Since SafeRefactor compares products' observable behavior by using testing, we measured the code coverage of the generated tests in each source product to evaluate their quality. In our experiment, the generated tests achieved low level of test coverage in Pairs 8, 12, and 33 (see Figure 4), which does not give much confidence in the tools' results. In fact, these tests did not find the behavioral changes introduced by the transformation, leading to false positives (except All Product Pairs in Pair 33, since it did not generate tests). In practice, developers can also use test coverage results to increase their confidence in our tools' results.

It is difficult, though, to establish what is a good statement coverage level to evaluate whether a transformation preserves behavior. Change coverage [Wloka et al., 2010] is another coverage metric that can be used to improve the confidence in the tools' results. This coverage metric evaluates whether the introduced changes were exercised by the test suite. Therefore, it does not matter whether the tests covered only 5% of the program, as long as these 5% were related to the methods impacted by the change. We plan to use this metric in future evaluations of our tools.

4.5 Answers to the Research Questions

From the evaluation results, we make the following observations:

– **Q1**. Do the approaches correctly classify the evolution scenarios?

  No. All Product Pairs does not identify behavior-preserving transformations when the sets of public method names are different between the given versions. In these cases, the other three approaches are more suitable to compare transformations, since they do not take into account the sets of public method names. Also, All Products, Impacted Products and Impacted Classes presented false positives in transformations involving UI components. We have similar results in other 15 evolution scenarios previously analyzed [Ferreira et al., 2012].

– **Q2**. Do the approaches have the same performance?

  No. Impacted Products and Impacted Classes are more efficient when analyzing transformations that only change the FM. Therefore, developers should use them in this scenario. Impacted Classes is faster when analyzing transformations that only change code, leading to reductions of up to 70% in time (Subject 7). The cost to run Impacted Classes increases as the number of changed classes grows, getting closer to the cost of running Impacted Products. So, developers should analyze this trade-off between precision and time in each situation. We also have similar results in our previous work [Ferreira et al., 2012], where the optimized approaches were more efficient in time for changes that affect only FM and CK. All Products and All Product Pairs are less efficient but they are more precise.

– **Q3**. Do the approaches have the same code coverage?

  No. However, considering the code coverage only in changed classes, the toolset has almost the same results for all pairs of commits analyzed. We do not evaluate this factor in our previous work [Ferreira et al., 2012].

Our results suggest that the proposed tools have similar accuracy (91%). However, All Product Pairs is slightly more precise (93% against 90% of the other ones), but has lower recall (96% against 100%). On the other hands, Impacted Products and Impacted Classes can reduce the time to check an SPL transformation. Therefore, we believe that developers should use All Product Pairs only if time is not a constraint, since the optimized tools have similar accuracy but can have much better performance.

4.6 Threats to Validity

With respect to construct validity, we created a baseline (see the column *Baseline* of Table 1) by manually analyzing the transformations and comparing the approaches' results since we did not previously know which versions contain safe evolution scenarios. However, manual analysis is an error prone activity.

There are some limitations related to SafeRefactor. For instance, it does not evaluate developer intention to refactor, but whether a transformation changes the product behavior. Moreover, in the closed world assumption, we have to use the test suite provided by the SPL that is being refined. SafeRefactor follows an open world assumption, in which every public method can be a potential target for the test suite generated by Randoop. However, this is often not the case considering the overall context of an SPL. So more precise results could be obtained with a tool that takes the context of specific SPLs into consideration.

With respect to internal validity, we use the same machine to test the subjects using the four tools to avoid influences in the measures. Besides we repeated five times the execution of each approach for each pair.

The FMs and CKs found in TaRGeT are implemented in a different format, incompatible with our tools. Early versions of its FM were implemented using Pure::Variants[3], and our tools are compatible only with FMs designed using Feature Modeling Plug-In [Czarnecki et al., 2004]. The CK format adopted in the latest versions of TaRGeT has a different semantics of the simple CK with just feature expressions mapped to assets that we use. It is compatible with the tool Hephaestus [Bonifácio et al., 2009] and allows feature expressions mapped to transformations like processing and copy files. However their operations are mappable to the simple CK semantics that we adopt. We manually converted those models, and to avoid mistakes we make sure that both models generate the same products with the same set of assets of the original ones.

Finally, the number of tests generated for each method used in SafeRefactor may have influence on the detection of non-refactorings.

With respect to external validity, we evaluated only one SPL (TaRGeT) due to the costs of manual analyses. Our results are not representative of all SPLs, but it corroborates with the results from our previous study where we applied 13 transformations in a different SPL (MobileMedia) [Ferreira et al., 2012].

5 Related Work

Alves et al. [Alves et al., 2006] informally present an SPL refactoring definition, based on FM changes that maintain or increase configurability. They propose FM transformations that conform to this definition. Borba [Borba, 2011] initially proposes the SPL refinement notion. He also illustrates different kinds of refinement transformation templates that can be useful for deriving and evolving SPLs. According to his definition, in an SPL refinement, the resulting SPL must be able to generate products that behaviorally match the original SPL products (not necessarily the same configurations). Since it is not needed to have the same set of product configurations in the resulting SPL, this definition allows feature renaming. We use this SPL refinement definition as basis for this work.

Borba et al. [Borba et al., 2010, Borba et al., 2012] propose an extended version of this previous formalization, where they explore a number of properties that justify stepwise and compositional evolution of product line artifacts. The properties are proven sound in a theorem prover. We base our optimized approaches in refinement properties presented in these works.

Thüm et al. [Thum et al., 2009] classify evolution of a FM in four categories: refactorings for changes that do not add or remove products; generalizations for

[3] Pure::Variants is a tool for variant management of product lines.

changes that add new products without removing the existing ones; specializations for changes that remove products without add new products; finally, arbitrary edits for other cases. The refinement notion used in our work is equivalent to their definitions for refactorings, generalizations and some cases of specializations, and is more comprehensive since it is based on the behavior preservation of the existing products taking into account not only changes in FM but also changes in CK and assets.

Czarnecki et al. [Czarnecki et al., 2005] introduce cardinality-based feature modeling. They specify a formal semantics for FMs and translate cardinality-based FMs into context-free grammars. They also propose FM specializations, a transformation that reduces configurability. Our approaches deal with FM specialization, but handle only the cases where all the source products have target products that behaviorally match with them [Borba et al., 2010].

Thaker et al. [Thaker et al., 2007] present techniques for verifying type safety properties of AHEAD [Batory, 2004] SPLs using FMs and SAT solvers. They extract properties from feature modules and verify that they hold for all SPL members. These properties are based on the AHEAD theory of program synthesis, and some of them do not reveal actual errors, but rather designs that *smell bad*. Similarly to this work, our well-formedness verification (Step 1) also extracts properties from the code assets, in terms of provided and required interfaces, and checks that they hold for all products from the FM. Also, our Alloy encoding provides sound and complete analysis, due to our scope being well delimited.

Early work [Critchlow et al., 2003] on SPL refactoring focuses on Product Line Architectures (PLAs) described in terms of high-level components and connectors. This work presents metrics for diagnosing structural problems in a PLA, and introduces a set of architectural refactorings that we can use to resolve these problems. Besides being specific to architectural assets, this work does not deal with other SPL artifacts such as FMs and CK. There is also no notion of behavior preservation for SPLs, as captured here by our SPL refinement notion.

A number of approaches [Kolb et al., 2005, Trujillo et al., 2006, Liu et al., 2006, Kastner et al., 2007] focus on refactoring a product into an SPL, not exploring SPL evolution in general, as we do here. Kolb et al. [Kolb et al., 2005] discuss a case study in refactoring legacy code components into an SPL. They define a systematic process for refactoring products with the aim of obtaining SPLs assets. There is no discussion about FMs and CK. As we do here, they check behavior preservation and configurability of the resulting SPLs by testing. Kastner et al. [Kastner et al., 2007] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs. As discussed here and elsewhere [Borba, 2011, Borba et al., 2010] these are not adequate for justifying SPL refinement and refactoring. Trujillo et al. [Trujillo et al., 2006] go beyond code assets, but do not explicitly consider

transformations to FM and CK. They also do not consider behavior preservation; they use the term "refinement", but in the different sense of overriding or adding extra behavior to assets.

Liu et al. [Liu et al., 2006] also focus on the process of decomposing a legacy application into features, but go further than the previously cited approaches by proposing a refactoring theory that explains how a feature can be automatically associated to a base asset (a code module, for instance) and related derivative assets, which contain feature declarations appropriate for different product configurations. Contrasting with our work, this theory does not consider FM transformations and assumes an implicit notion of CK based on the idea of derivatives. Also, our focus was on SPL transformations, instead of refactoring single programs into SPLs.

Kim et al. [Kim et al., 2011] explore the concept of irrelevant features to reduce SPL testing. These features do not have impact on the tests. They aim at pruning the space of such features to reduce the number of SPL programs to examine for that test without reducing its ability to find bugs. Their work does not focus on proposing a tool for checking SPL refinement. Our tools evaluate a transformation using SafeRefactor. They analyze a transformation considering FM and CK optimizations and generate tests. We can use their results and improve it by avoiding generating insignificant tests in order to optimize our tools. Similarly, our Impacted Products tool avoids the combinatorial number of products by not evaluating products that are not affected by a change.

Soares et al. [Soares et al., 2008] propose an SPL variability refactoring tool (FLiP) based on the Eclipse plugin platform to perform source code refactorings to extract product variations. This tool focuses on refactoring templates using AspectJ that can change the CK and code. However, it has a limited set of refactoring templates, does not automatically transform FM and it allows users to choose transformations without checking any refactoring rules. Moreover, it does not check behavior preservation after changes. Previous works demonstrate that automatic refactorings are susceptible to bugs [Soares et al., 2010]. We believe this tool is complementary to our approaches, since we could check if FLiP transformations are SPL refinements indeed.

Neves et al. [Neves et al., 2011] analyze product line evolutions and described safe evolutions templates that developers can use when evolving product lines. We propose a more general approach, that can improve the confidence that a transformation is safe, regardless the template of the transformation. Our tools could complement the usage of these templates, checking behavior preservation.

Gheyi et al. [Gheyi et al., 2011] propose a set of FM refactorings, and an automatic approach for checking whether a general or specific FM transformation is a refactoring. We use their approach encoding FM in Alloy to check refactorings in FMs. However our checkers are more comprehensive since they consider

more product line artifacts as CK and asset mapping.

6 Conclusions

In our previous work [Ferreira et al., 2012], we propose four tools for checking whether SPL evolution scenarios are refinements. We implemented them based on a formal SPL refinement notion proposed by Borba et al. [Borba et al., 2012]. The suitability of each tool depends on the kind of change and on user's constraints regarding time and reliability. In this article, we formalized the algorithms of these tools and analyzed them in 35 evolution scenarios applied by developers to a real SPL (32 KLOC). We compare them with respect to soundness, performance and test coverage.

As future work, we plan to evaluate our tools in other case studies. Moreover, we aim at proposing new optimizations based on other behavioral preservation properties [Borba et al., 2012]. Additionally, we can improve performance of our tools by using incremental compilation and parallelism.

Acknowledgment

We gratefully thank the anonymous referees from SBCARS and J.UCS for their useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES).

References

[Alves et al., 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring product lines. In *GPCE*, pages 201–210.

[Batory, 2004] Batory, D. (2004). Feature-oriented programming and the AHEAD tool suite. In *ICSE*, pages 702–703.

[Bonifácio et al., 2009] Bonifácio, R., Teixeira, L., and Borba, P. (2009). Hephaestus a tool for managing spl variabilities. In *SBCARS*, Natal, Brazil.

[Borba, 2011] Borba, P. (2011). An introduction to software product line refactoring. In *GTTSE*, volume 6491 of *Lecture Notes in Computer Science*, pages 1–26. Springer.

[Borba et al., 2010] Borba, P., Teixeira, L., and Gheyi, R. (2010). A theory of software product line refinement. In *ICTAC*, volume 6255 of *LNCS*, pages 15–43. Springer.

[Borba et al., 2012] Borba, P., Teixeira, L., and Gheyi, R. (2012). A theory of software product line refinement. *Theoretical Computer Science*, 455:2–30.

[Critchlow et al., 2003] Critchlow, M., Dodd, K., Chou, J., and van der Hoek, A. (2003). Refactoring product line architectures. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, and Effects*, pages 23–26.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. (2000). *Generative programming: methods, tools, and applications*. Addison-Wesley.

[Czarnecki et al., 2005] Czarnecki, K., Helsen, S., and Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29.

[Czarnecki et al., 2004] Czarnecki, K., Helsen, S., and Ulrich, E. (2004). Staged configuration using feature models. In *SPLC*, volume 3154 of *LNCS*, pages 266–283.

[Ferreira et al., 2012] Ferreira, F., Borba, P., Soares, G., and Gheyi, R. (2012). Making software product line evolution safer. In *SBCARS*, pages 21–30.

[Ferreira et al., 2010] Ferreira, F., Neves, L., Silva, M., and Borba, P. (2010). TaRGeT: a model based product line testing tool. In *CBSoft – Tools Session*, CBSoft '10.

[Figueiredo et al., 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., and Dantas, F. (2008). Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, pages 261–270.

[Gheyi et al., 2006] Gheyi, R., Massoni, T., and Borba, P. (2006). A theory for feature models in Alloy. In *Proceedings of the 1st Alloy Workshop*, pages 71–80.

[Gheyi et al., 2011] Gheyi, R., Massoni, T., and Borba, P. (2011). Automatically checking feature model refactorings. *J. UCS*, 17(5):684–711.

[Jackson, 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. MIT press.

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI CMU.

[Kastner et al., 2007] Kastner, C., Apel, S., and Batory, D. (2007). A case study implementing features using AspectJ. In *SPLC*, pages 223–232.

[Kim et al., 2011] Kim, C. H. P., Batory, D. S., and Khurshid, S. (2011). Reducing combinatorics in testing product lines. In *AOSD*, pages 57–68.

[Kolb et al., 2005] Kolb, R., Muthig, D., Patzke, T., and Yamauchi, K. (2005). A case study in refactoring a legacy component for reuse in a product line. In *ICSM*, pages 369–378.

[Liu et al., 2006] Liu, J., Batory, D., and Lengauer, C. (2006). Feature oriented refactoring of legacy applications. In *ICSE*, pages 112–121.

[Mongiovi et al., 2014] Mongiovi, M., Gheyi, R., Soares, G., Teixeira, L., and Borba, P. (2014). Making refactoring safer through impact analysis. *Science of Computer Programming*.

[Neves et al., 2011] Neves, L., Teixeira, L., Sena, D., Alves, V., Kulezsa, U., and Borba, P. (2011). Investigating the safe evolution of software product lines. In *GPCE*, pages 33–42.

[Opdyke, 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, UIUC.

[Pacheco et al., 2007] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *ICSE*, pages 75–84.

[Pohl et al., 2005] Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[Ren et al., 2004] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. (2004). Chianti: a tool for change impact analysis of Java programs. In *OOPSLA*, pages 432–448.

[Soares et al., 2013] Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE TSE*, 39(2):147–162.

[Soares et al., 2010] Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, 27:52–57.

[Soares et al., 2008] Soares, S., Calheiros, F., Nepomuceno, V., Menezes, A., Borba, P., and Alves, V. (2008). Supporting software product lines development: FLiP - product line derivation tool. In *SPLASH*, pages 737–738.

[Teixeira et al., 2013] Teixeira, L., Borba, P., and Gheyi, R. (2013). Safe composition of configuration knowledge-based software product lines. *JSS*, 86(4):1038–1053.

[Thaker et al., 2007] Thaker, S., Batory, D., Kitchin, D., and Cook, W. (2007). Safe composition of product lines. In *GPCE*, pages 95–104.

[Thum et al., 2009] Thum, T., Batory, D., and Kastner, C. (2009). Reasoning about edits to feature models. In *ICSE*, pages 254–264.

[Trujillo et al., 2006] Trujillo, S., Batory, D., and Diaz, O. (2006). Feature refactoring a multi-representation program into a product line. In *GPCE*, pages 191–200.

[van der Linden et al., 2007] van der Linden, F., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Springer.

[Wloka et al., 2010] Wloka, J., Hoest, E., and Ryder, B. G. (2010). Tool support for change-centric test development. *IEEE Software*, 27(3):66–71.