Making Software Product Line Evolution Safer

Felype Ferreira*, Paulo Borba*, Gustavo Soares†, Rohit Gheyi†
*Informatics Center, Federal University of Pernambuco
Email: {fsf2, phmb}@cin.ufpe.br
†Department of Computing Systems, Federal University of Campina Grande
Email: {gsoares, rohit}@dsc.ufcg.edu.br

***Abstract*—Developers evolve software product lines (SPLs) manually or using typical program refactoring tools. However, when evolving a product line 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, in a SPL we typically have to deal with a set of possibly alternative assets that do not constitute a well-formed program. As a result, manual changes and existing program refactoring tools may introduce behavioral changes or invalidate existing product configurations. To avoid that, we propose approaches and implement tools for making product line evolution safer; these tools check whether SPL transformations are refinements in the sense that they preserve the behavior of the original SPL products. They implement different and practical approximations of a formal definition of SPL refinement. We evaluate the approaches in concrete SPL evolution scenarios where existing product's behavior must be preserved. However, our tools found that some transformations introduced behavioral changes. Moreover, we evaluate defective refinements, and the toolset detects the behavioral changes.**

***Index Terms*—software product lines, product line evolution, checking tools, refactoring, refinement, safe evolution.**

I. Introduction

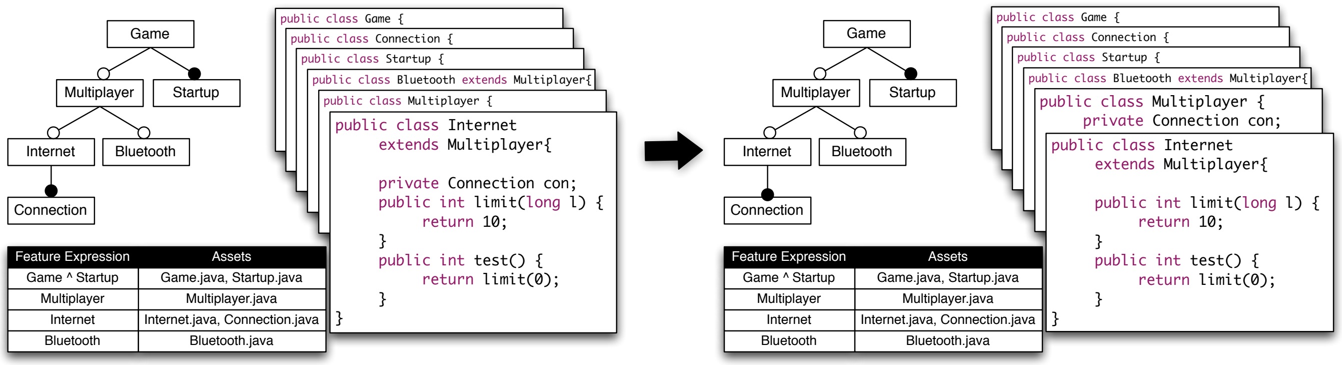
A software product line (SPL) is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [1], [2]. 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, like feature models [3] and configuration knowledge [4], that enable product generation, and they should all be changed consistently.

In particular, often during SPL evolution, it might be important to make sure that the associated changes do not affect the behavior of the existing SPL products. We need that when refactoring a SPL, that is, simply improving the design of its artifacts. We also need that when extending a SPL by making it able to generate new products, including new optional features, for example. This notion of safe evolution is captured by a formal notion of SPL refinement [5], [6], which guarantees that the observable behavior of products in the original SPL is preserved by corresponding products in the new, evolved, SPL. An associated catalogue of safe SPL evolution transformations [7] could be used by developers, in the same way that object-oriented single program refactoring catalogues are available in current development environments. However, in many practical contexts, catalogue driven evolution is not appealing.

So 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 such as the ones we find in Eclipse and NetBeans. These tools check a number of preconditions for behavior preservation of the modified assets. But they not only may perform incorrect refactorings for single programs [8], 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 feature model (FM) and configuration knowledge (CK), which should be consistently changed together with the reusable assets. As a result, supposedly safe evolution scenarios turn up to be unsafe, inappropriately changing the behavior of existing products and negatively impacting users and development productivity. 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.

To help to solve this problem, in this work we propose and evaluate four testing based approaches and their implementations (a toolset) for checking SPL refinement, and therefore safe SPL evolution. As a basis for the implementations, we use the formal definition of SPL refinement [5], [6] we mentioned before—in summary, we basically try to aproximate checking that the resulting, evolved, SPL is able to generate products that behaviorally match the original SPL products. The toolset compares two versions of a SPL, before (source) and after (target) the changes, using automatically generated tests to evaluate if observable behavior of the source products match observable behavior of associated target products. Moreover, our tools also check whether all products in the target version are well-formed. To compare the behavior of products before and after the evolution scenarios, we use an adapted version of Safe Refactor [8], a tool for detecting behavioral changes. Our toolset consists of four tools named after the approaches they implement: All product pairs, All products, Impacted Products, and Impacted Classes. 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.

In practice, checking whether the target SPL generates



Feature Expression	Assets
Game ^ Startup	Game.java, Startup.java
Multiplayer	Multiplayer.java
Internet	Internet.java, Connection.java
Bluetooth	Bluetooth.java

```
public class Game {
public class Connection {
public class Startup {
public class Bluetooth extends Multiplayer{
public class Multiplayer {
public class Internet
      extends Multiplayer{

      private Connection con;
      public int limit(long l) {
            return 10;
      }
      public int test() {
            return limit(0);
      }
}
```

Feature Expression	Assets
Game ^ Startup	Game.java, Startup.java
Multiplayer	Multiplayer.java
Internet	Internet.java, Connection.java
Bluetooth	Bluetooth.java

```
public class Game {
public class Connection {
public class Startup {
public class Bluetooth extends Multiplayer{
public class Multiplayer {
     private Connection con;
public class Internet
      extends Multiplayer{

      public int limit(long l) {
            return 10;
      }
      public int test() {
            return limit(0);
      }
}
```

Fig. 1: SPL evolution yields sets of assets that do not represent well-formed products.

products with compatible behavior of the original ones may be overly time consuming in SPLs with a large number of possible combinations. The ALL PRODUCT PAIRS, instead of checking for each source product if one of the target products behaviorally matches with it, the tool first checks if each source product has compatible behavior with the target product with the same configuration of it. Exceptionally when the products' behaviors do not match or when source product's configuration does not exist in the target SPL, the ALL PRODUCT PAIRS tool compares the source product's behavior against all the other target products. The ALL PRODUCTS tool is similar to the ALL PRODUCT PAIRS, however it only compares source products with target products with the same configuration, simplifying the checking.

The IMPACTED PRODUCTS and the IMPACTED CLASSES tools contain further optimizations to reduce even more the costs of checking SPL refinements. They are based on behavioral preservation properties [5] that allow optimizing the evaluation of certain kinds of evolution scenarios. The former 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. The latter, instead of checking products, focuses on testing only the changed assets. This way, it avoids generating and testing all impacted products, which can lead to a major reduction on time compared to the first two tools.

We evaluate[1] our tools in 15 evolution scenarios of two SPLs ranging up to 32 KLOC. The existing product's behavior should have been preserved in 10 evolution scenarios of the TaRGeT SPL, a tool that automatically generates functional tests from use case documents written in natural language [9]. However, we found that changes in 2 of them erroneously introduced behavioral changes. Besides, we evaluate our tools with a set of 5 defective refinements performed to SPL examples and the MobileMedia SPL [10], an SPL for applications that manipulates music, video and photo on mobile devices. Our tools automatically detected all behavioral changes. The main contributions of this work are the following:

[1]All experiment data are available at: http://www.cin.ufpe.br/~fsf2/sbcars_experiments.html

- Four tools to evaluate whether a SPL evolution scenario is a refinement (Section III);
- An evaluation of our tools, with respect to their performance and effectiveness for checking SPL refinement, in 10 evolution scenarios to refine the TaRGeT SPL, and in 5 defective SPL refinements, 3 in MobileMedia and 2 in SPL examples (Section IV).

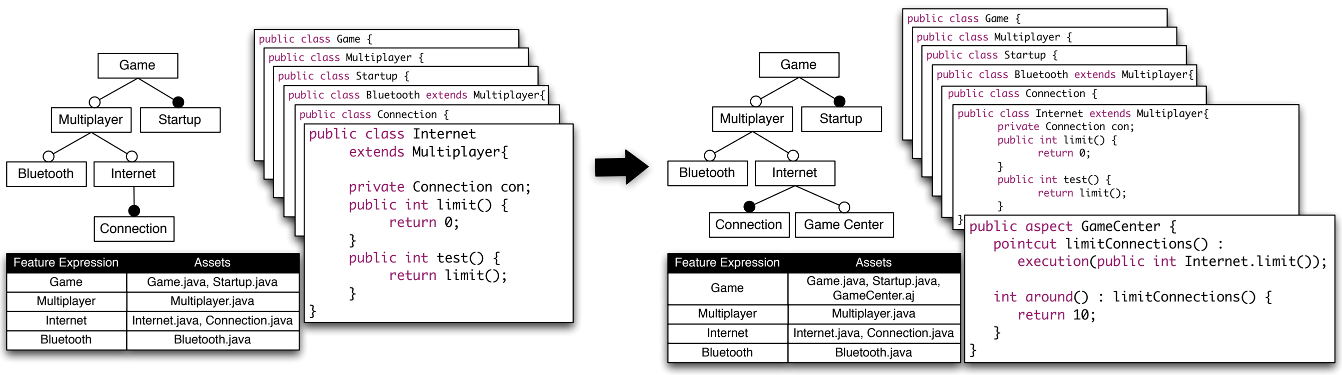
The remaining of this paper is organized as follows. In Section II, we show problems that may happen while evolving SPLs. In Section III, we describe our tools and their implementations. Next, we evaluate them in Section IV. Finally, we present related work and final remarks in Sections V and VI, respectively.

II. MOTIVATING EXAMPLES

When evolving an SPL, developers often manually change the different SPL artifacts like FMs [3] and reusable assets. To change the code assets, they might also use code refactoring tools such as the one provided by Eclipse. Unfortunately, this can lead to problems like the generation of non well-formed products or undesirable changes to the behavior of the existing ones. For example, consider the simple SPL evolution scenario with a toy example of game SPL that simplifies real problems that we found during the analysis of larger SPLs.

Figure 1 depicts its 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). The figure also shows the code assets [4] and the CK, which is responsible for driving product generation. It 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. For example, 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`}.



Feature Expression	Assets
Game	Game.java, Startup.java
Multiplayer	Multiplayer.java
Internet	Internet.java, Connection.java
Bluetooth	Bluetooth.java

```
public class Game {
public class Multiplayer {
public class Startup {
public class Bluetooth extends Multiplayer{
public class Connection {
public class Internet
      extends Multiplayer{

      private Connection con;
      public int limit() {
            return 0;
      }
      public int test() {
            return limit();
      }
}
```

Feature Expression	Assets
Game	Game.java, Startup.java, GameCenter.aj
Multiplayer	Multiplayer.java
Internet	Internet.java, Connection.java
Bluetooth	Bluetooth.java

```
public class Game {
public class Multiplayer {
public class Startup {
public class Bluetooth extends Multiplayer{
public class Connection {
public class Internet extends Multiplayer{
      private Connection con;
      public int limit() {
            return 0;
      }
      public int test() {
            return limit();
      }
}
```

```
public aspect GameCenter {
   pointcut limitConnections() :
      execution(public int Internet.limit());

   int around() : limitConnections() {
      return 10;
   }
}
```

Fig. 2: Add feature evolution introduces behavioral changes in existing products.

Besides FM and CK, an SPL also has 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.

A. Dependences

To illustrate, suppose we apply the *Pull up field* refactoring to move the field `Internet.con` to the `Multiplayer` class. Eclipse performs the change without any warning. It is not aware of the other artifacts, FM and CK, nor to the fact that some of the assets might conflict. For example, they might be alternative implementations of the same concept; so the set of assets of a SPL might not even constitute a valid program. As Eclipse is oblivious to that, the evolution scenario is carried out only changing the mentioned classes, as illustrated in Figure 1, with the resulting `Multiplayer` and `Internet` classes, which still compile after this evolution.

However, this evolution step is not safe; it is not a SPL refinement because the resulting SPL will generate invalid products, sets of assets that do not compile. For instance, 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, a 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`.

The same kind of problem might also occur when manually applying refactorings to code assets, or even when trying to improve the CK and FM. In summary, presumably safe modifications to SPL artifacts might turn up to be unsafe modifications for the SPL as a whole.

B. 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, 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 shows the manually applied changes.

In this case, all products of the resulting SPL are well-formed. However, the evolution does not preserve the behavior of the existing products and is not safe. 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.

A similar problem happened during the development of the TaRGeT SPL, a tool that automatically generates functional tests from use case documents written in natural language [9]. At that time, the tool was able to generate tests in four different output formats. The development team performed an evolution to add a new output option and manually changed SPL artifacts to add the assets for the new `TestLink` feature. However, they specified an incorrect association in the CK. Instead of associating the new assets with the new feature, they associated them with the existing `XML` feature. So, the behavior of the new assets was added to all products that contain the `XML` feature. Therefore, products without the new feature experienced an unexpected behavior because the `TestLink` feature functionality was accidentally added to them.

Similar issues can also appear when changing only assets or FMs, or the three kinds of artifcats at the same time. For instance, we might forget to include an entry in the CK, include incompatible entries in it, or make wrong associations between features and assets likewise in our toy example. The previous examples are small and developers could maybe detect, without tool support, the issues we have discussed. However, efficiently checking the introduction of compilation

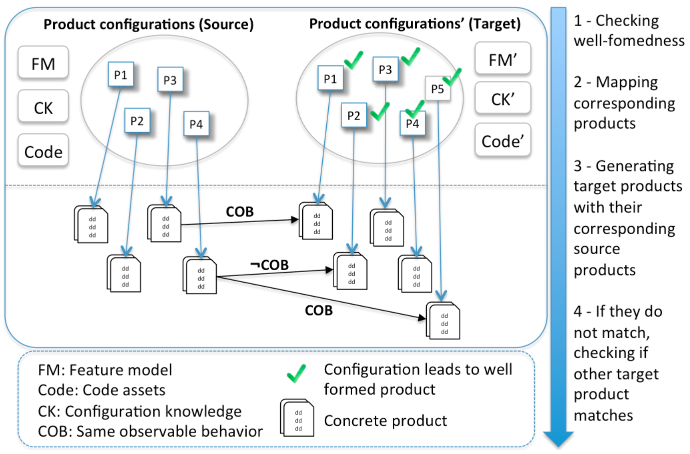

Product configurations (Source)
Product configurations' (Target)
FM
CK
Code
FM'
CK'
Code'
P1
P2
P3
P4
P5
COB
¬COB
COB
1 - Checking well-fomedness
2 - Mapping corresponding products
3 - Generating target products with their corresponding source products
4 - If they do not match, checking if other target product matches
FM: Feature model
Code: Code assets
CK: Configuration knowledge
COB: Same observable behavior
Configuration leads to well formed product
Concrete product

Fig. 3: The ALL PRODUCT PAIRS tool.

errors and behavioral changes in industrial SPLs that contain a number of products may be more difficult. This can decrease SPL development productivity. Therefore, we need better tool support for evolving SPLs. In this paper, we propose a toolset for making SPL evolution safer. We focus on evolutions that intend to preserve observable behavior.

III. Tool Support for Checking SPL Refinements

In this section, we describe our toolset for checking SPL safe evolution. We propose four tools, named after the approaches they implement and based on the SPL refinement definition. In the following sections, we describe each tool and further details about their implementation.

A. All product pairs

Following the basic idea of SPL refinement [5] to ensure the safe evolution of a SPL, we basically have to check that the resulting SPL must be able to generate products that behaviorally match the original SPL products, presenting compatible observable bahavior. This is enough to guarantee that design changes or the introduction of new features and products do not impact the users of the original SPL.

ALL PRODUCT PAIRS is our baseline tool for checking SPL refinements. It tries to directly check the condition expressed by the SPL refinement definition, looking for corresponding products with compatible observable behavior, which are products that behaviorally match when we compare them, as illustrated in Figure 3.

The tool first checks if the target SPL is well-formed (Step 1), which means that it still generates well-formed products, that correspond to valid products in the underlying languages used to describe assets [5]. If it finds a problem, it stops the process, reports all the invalid product configurations found and indicates that the SPL is not refined and the evolution is not safe. If it does not find a problem, for each product in the source SPL, it analyzes if there is a product with the same configuration in the target SPL and maps it as the likely corresponding target product (Step 2). After mapping products, it checks whether each product in the source SPL and its likely corresponding, when it exists, have compatible observable behavior using randomly generated unit test cases (Step 3). Exceptionally, when their observable behavior is not compatible or the source configuration does not exist in the target SPL, the tool compares the behavior of the source product against all the other target products until finding its corresponding or exhausting the possibilities. (Step 4). If it does not find any corresponding product in the target SPL, it assumes that the SPL is not refined, reporting the first occurrence of product configuration without corresponding refined product, and the set of tests that reveal 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 a SPL refinement and is a safe evolution. This is an approximation of behavioral preservation as defined in the SPL refinement definition since tests cannot prove the absence of behavioral changes. A full guarantee cannot often be given 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.

To illustrate it, consider the motivating examples in Section II. For the first example (Section II-A), 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 II-B), it does not detect problems in Step 1. Then, for each of the five products in the source SPL it analyzes whether corresponding products exist 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 assets) 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. Then, in Step 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 not safe.

To use SAFE REFACTOR to compare SPL products behavior, we changed its analysis. Its original version identifies the common methods between source and target programs and compares their behavior with respect to these methods. 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 the features only presented in one of the versions, which may lead to false positives: the tool reports a refinement but source and target products have different behavior. Therefore, during the analysis, when the tool detects that the source and target products have different public methods, it considers that they do not have compatible 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 proceeds the evaluation of the products.

Although this change avoids false positives, it can generate false negatives, that is, the tool reports that the source and target products do not have compatible behavior, but they do have, when public methods are removed or added. We address this issue in the next tool.

B. ALL PRODUCTS

As the ALL PRODUCT PAIRS does not compare the products behavior when they have a different set of public methods, it may lead to false negatives. To evaluate how often these cases may happen, we have the ALL PRODUCTS tool.

It is very similar to ALL PRODUCT PAIRS, however it has a difference in Step 4. 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. As 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, during the evolution, developers add or remove public methods. 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, when some source product has a corresponding product in the target SPL but source and target product has different sets of assets. We discuss more about this in Section IV-D.

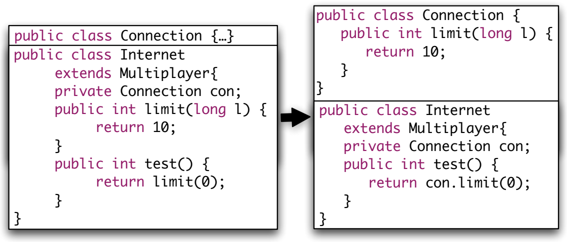
As the ALL PRODUCTS can only compare source and target products with the same sets of assets, if some source product does not have a likelly corresponding product in the target SPL, ALL PRODUCTS is not able to be applied.

C. Optimized Approaches

This section presents tools with further optimizations to reduce the cost for checking SPL refinement of the previous tools. We analyze the source and target SPLs and use refinement properties [5] 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 and just 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) [11].

If this condition is not satisfied, we can only apply ALL PRODUCT PAIRS (APP) 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).



```
public class Connection {…}
public class Internet
      extends Multiplayer{
      private Connection con;
      public int limit(long l) {
            return 10;
      }
      public int test() {
            return limit(0);
      }
}
```

```
public class Connection {
   public int limit(long l) {
      return 10;
   }
}
public class Internet
   extends Multiplayer{
   private Connection con;
   public int test() {
      return con.limit(0);
   }
}
```

Fig. 4: *Move method* refactoring.

For instance, consider the FM of our motivating example (see Figure 1). The `Startup` feature is mandatory. Suppose we change it to optional. Since it just changes the FM, we can perform Steps 1 and 2 only. They detect that the resulting sets of assets contain the original ones. Thus, as code did not change, the tools assume that the evolution scenario is safe.

1) IMPACTED PRODUCTS*:* 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. First, it identifies the code assets modified by the evolution. By checking which product configurations contain these asset names, it identifies the products impacted by the change. Then, it only generates and evaluates source products containing at least one changed asset.

For instance, imagine that instead of applying the *Pull up field* refactoring to our first motivating example (see Section II), we applied a *Move method* refactoring to `Internet.limit(long)`, moving it to the `Connection` class. Figure 4 shows the *Move Method* evolution scenario applied to our motivating example (see Section II). By comparing the source and target versions of the assets, our tool identifies that classes `Internet` and `Connection` are modified. The `limit(long)` method is moved from `Internet` to `Connection`. The tool then analyzes the possible product configurations using the CK and FM. From Figure 4, we see modified classes are related to the products {`Game`, `Multiplayer`, `Startup`, `Internet`, `Connection`} and {`Game`, `Multiplayer`, `Startup`, `Internet`, `Connection`, `Bluetooth`}. So, these are the impacted products for this evolution scenario. The IMPACTED PRODUCTS tool then, differently from ALL PRODUCT PAIRS, only generates products that contain at least one changed asset. In this example, the optimization reduces the number of evaluated source products from 5 to 2.

The IMPACTED PRODUCTS tool, in the worst case, checks as many products as ALL PRODUCTS. This can happen when changed assets are related to the root feature, for example.

2) IMPACTED CLASSES*:* 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 [5], 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, the evaluation using IMPACTED CLASSES tends to be faster.

First, the IMPACTED CLASSES tool identifies modified assets. For each one, the tool computes its dependences, that is, the set of other assets needed to compile the modified asset. We call this set of modified assets with their dependences as *sub products*. Our tool compiles the source and target versions of each sub product. It then checks, for each sub product, whether they have compatible observable behavior, generating test only for changed classed.

For instance, consider the same *Move Method* refactoring used to illustrate the previous tool (see Section III-C1). 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 reasoning, the tool computes dependences for each identified dependence. For this class, the tool generates the sub product consisting on 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 stress that although costly, we can use ALL PRODUCT PAIRS to check any kind of evolution scenarios, while optimized tools are suitable only when FM and CK are refined [11]. Moreover, with IMPACTED CLASSES we may lose precision, since local changes in OO classes may indirectly impact other ones, and this tool just focuses on changed classes without taking into account all the contexts where it is used in the SPL.

D. Implementation

In this section, we describe technical details of our implementation. First, we explain the ALL PRODUCT PAIRS and ALL PRODUCTS tools, and then the optimized ones.

To perform Step 1 of our first tool, we use Alloy, a formal specification language, with a specific theory for FMs [12]. The tool support of Alloy (Alloy Analyzer) enables us to perform automatic analysis on Alloy models. Our tools translate the FM and CK into propositional Alloy formulae following the encoding proposed in previous work [12], using an extension of the existing CK model expressing dependences between assets [13]. We could use SAT solvers directly to improve our results, however we use Alloy because it easily work with any different SAT solver, independently of a specific implementation.

Step 2 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 (Steps 3 and 4), we use SAFE REFACTOR [8].

When optimized tools are applicable, they identify the set of changed assets. We implemented an Abstract Syntax Tree (AST) comparator to compare each version of the code assets.

Differently from IMPACTED PRODUCTS and ALL PRODUCT PAIRS, the IMPACTED CLASSES tool does not check whole products. Instead, after identifying the changed assets using the AST Comparator, it uses the Soot framework[2] for computing their dependences. The union of the changed asset and its dependences generates a sub product, that is a minimal set of classes that can be compiled to perform tests that SAFE REFACTOR uses to evaluate them. 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.

Additionally, for the IMPACTED CLASSES tool, we give special attention to SPLs implemented with conditional compilation, which need pre-processing to obtain valid classes. To deal with conditional compilation, 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 SAFE REFACTOR 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 SAFE REFACTOR for checking behavioral changes with each of them.

IV. EVALUATION

In this section, we present the evaluation of the approaches implemented by our toolset for checking SPL refinement. We evaluated them in 15 evolution scenarios applied to SPLs ranging up to 32 KLOC and more than a thousand possible product configurations. First, we characterize the subjects in Section IV-A. In Section IV-B, we describe the experimental setup. Finally, in Sections IV-C, IV-D and IV-E, we show the results, discuss related issues and threats to validity.

A. Subject characterization

We evaluate two categories of subjects. All of them consisting of SPL evolution scenarios. The first category consists of unsafe evolution scenarios we performed to introduce behavioral changes in an SPL. We apply our tools to these scenarios to make sure they are able to identify the unsafe modificiations to a SPL. The second category consists of supposedly safe evolution scenarios; the SPL developers believe these are safe evolution scenarios, and we use our tools to compare the four tools with respect to their performance and effectiveness for checking SPL refinement. In particular, we want to evaluate if they are able to correctly identify unsafe evolution scenarios.

Table I shows the subjects (pairs of source and target SPLs) used in the experiment in the previously explained categories: *Catalog of defective refinements* and *Real evolution scenarios applied to TaRGeT SPL*. Each of them is uniquely identified (Column *Subject*). Columns *KLOC* and *Features* show the size in lines of code and number of features of

[2]Soot is a Java optimization framework that allows analyzing Java bytecode. http://www.sable.mcgill.ca/soot/

Catalog of defective refinements

Subject	SPL	KLOC	Features	Products	Changed Artifacts	Description
1	Motivating Example 1	0.03 → 0.03	6 → 6	8 → 8	Code	Invalid product configurations
2	Motivating Example 2	0.03 → 0.03	6 → 6	8 → 8	Code	Behavioral change in 1 class
3	MobileMedia v3′	1.36 → 1.37	8 → 8	9 → 9	Code	Behavioral change in 2 classes
4	MobileMedia v4′	1.56 → 1.56	9 → 9	21 → 21	Code	Behavioral change in 2 classes
5	MobileMedia v4′′	1.56 → 1.56	9 → 9	21 → 21	Code	Behavioral change in 5 classes

Real evolution scenarios to TaRGeT SPL

Subject	SPL	KLOC	Features	Products	Changed Artifacts	Description
6	TaRGeT	32.54 → 32.54	22 → 22	1008 → 1512	FM	Replace alternative
7	TaRGeT	20.79 → 20.76	12 → 12	12 → 12	Code	Removal of warnings
8	TaRGeT	14.64 → 14.82	9 → 9	1 → 1	FM, CK, code	Extract feature Output
9	TaRGeT	14.82 → 15.17	9 → 10	1 → 2	FM, CK, code	Add feature TC4 Output
10	TaRGeT	28.13 → 28.37	13 → 14	24 → 36	FM ,CK, code	Add feature HTML Output
11	TaRGeT	28.67 → 29.07	16 → 17	48 → 60	FM, CK, code	Add feature XML TestLink Output
12	TaRGeT	29.76 → 30.34	18 → 19	140 → 168	FM, CK, code	Add feature STD Output
13	TaRGeT	28.51 → 28.62	14 → 15	36 → 48	FM, CK, code	Add feature XML Output
14	TaRGeT	18.63 → 22.04	12 → 13	12 → 24	FM, CK, code	Add feature CNL
15	TaRGeT	15.30 → 20.21	10 → 11	2 → 4	FM, CK, code	Add feature CM

TABLE I: Summary of subjects evaluated in the experiment; → illustrates values before and after the evolution scenarios; KLOC = thousands lines of code; Features = number of features; Products = number of product configurations.

each SPL, respectively. Column *Products* indicates the total of possible product configurations. Columns *Changed Artifacts* and *Description* describe the kind of change made to the SPL.

In the first category, we analyze MobileMedia [10], an SPL for applications that manipulates music, video and photo on mobile devices, and our toy examples. For Subjects 1 to 5, we introduce behavioral changes in MobileMedia OO releases and toy examples. In the second category, we analyze the TaRGeT SPL [9], a tool that automatically generates functional tests from use case documents written in natural language. It has more than 32 KLOC on its last release and it is an Eclipse Rich Client Platform (RCP) application. Based on comments in its SVN revisions, TaRGeT's developers selected evolutions scenarios that intended to be safe. Subject 6 consists of a FM evolution that turned an alternative relation into an or, increasing possible configurations. Subject 7 is a code transformation that intended to remove compiler warnings. Subjects 8-15 consist of transformations for extracting and adding features. In these subjects, developers jointly transformed FM, CK, and code.

B. *Experimental setup*

We ran our experiment on a quad-processor 2.66-GHz Server with 8 GB of RAM running Ubuntu 10.04. We defined a maximum number of tests to generate based on the number of methods to test for each pair of products, generating 2 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 SAFE REFACTOR [14].

To generate TaRGeT products, we use the FM and CK available from its SVN history. We use the MobileMedia FM from its documentation, and we systematically translated its CK implementation from the build files. In OO releases, conditional compilation directives represent the CK.

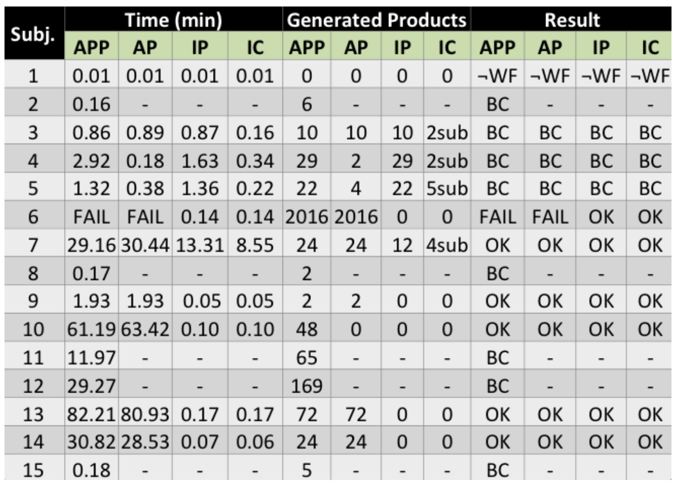
C. *Results*

The four tools associated to our four approaches detected all behavioral changes in the first category. Moreover, in the second category of subjects, although the SPL developers believed the transformations were refinements, our tools detect that two out of the ten analyzed transformations introduced behavioral changes. This shows some evidence that our tools can help SPL developers early detect unsafe evolution steps.

Table II shows the summary of our experimental results. Each line corresponds to a subject. Columns show the evaluated subject, the total execution time in minutes, the number of evaluated products, and the results of each tool. As we discuss in Section III-C2, we can only use IMPACTED CLASSES (IC), IMPACTED PRODUCTS (IP) and ALL PRODUCTS (AP) in transformations that preserve sets of assets. Therefore, some cells of the table have ”-”.

In the first category of subjects, our tools quickly detected the compilation error introduced in some products in Subject 1 by checking safe composition of the target SPL (Step 1). In this subject, the target SPL produces invalid products because changes applied to the assets were not updated in the CK. Subjects 3-5 show transformations where all tools detected the behavioral changes. While the IMPACTED CLASSES tool was the fastest one, the IMPACTED PRODUCTS tool was slower as ALL PRODUCT PAIRS. This happens because the changes were made in classes associated to all the procuts, leading IMPACTED PRODUCTS to evaluate as many products as ALL PRODUCT PAIRS. Since IMPACTED PRODUCTS also spends time identifying which classes have changed, it took more time to perform it than ALL PRODUCT PAIRS.

In the second category, the ALL PRODUCT PAIRS and ALL PRODUCTS tools failed to analyze Subject 6. To check this subject, it is necessary to generate and evaluate more than two thousand products. This resulted in an out of memory error, confirming the need for optimized tools. The IMPACTED



Subj.	Time (min)				Generated Products				Result			
	APP	AP	IP	IC	APP	AP	IP	IC	APP	AP	IP	IC
1	0.01	0.01	0.01	0.01	0	0	0	0	¬WF	¬WF	¬WF	¬WF
2	0.16	-	-	-	6	-	-	-	BC	-	-	-
3	0.86	0.89	0.87	0.16	10	10	10	2sub	BC	BC	BC	BC
4	2.92	0.18	1.63	0.34	29	2	29	2sub	BC	BC	BC	BC
5	1.32	0.38	1.36	0.22	22	4	22	5sub	BC	BC	BC	BC
6	FAIL	FAIL	0.14	0.14	2016	2016	0	0	FAIL	FAIL	OK	OK
7	29.16	30.44	13.31	8.55	24	24	12	4sub	OK	OK	OK	OK
8	0.17	-	-	-	2	-	-	-	BC	-	-	-
9	1.93	1.93	0.05	0.05	2	2	0	0	OK	OK	OK	OK
10	61.19	63.42	0.10	0.10	48	0	0	0	OK	OK	OK	OK
11	11.97	-	-	-	65	-	-	-	BC	-	-	-
12	29.27	-	-	-	169	-	-	-	BC	-	-	-
13	82.21	80.93	0.17	0.17	72	72	0	0	OK	OK	OK	OK
14	30.82	28.53	0.07	0.06	24	24	0	0	OK	OK	OK	OK
15	0.18	-	-	-	5	-	-	-	BC	-	-	-

TABLE II: Summary of experimental results; Time = total execution time in minutes; Generated Products = number of products generated and evaluated; sub = sub product; ¬WF = Not well-formed sets; BC = behavior change; FAIL = Fail during the execution of the tool.

PRODUCTS (IP) and IMPACTED CLASSES (IC) tools successfully performed the evaluation in less than a minute. Since the evolution scenario only involve changes to the FM, these tools only perform Steps 1 and 2 to check if it is an SPL refinement.

In Subjects 8 and 15, ALL PRODUCT PAIRS states that the evolutions are not refinements. However, by manually analyzing them, we notice developers introduced new public methods related to optional features in mandatory assets but these methods are only called in the programs when these optional features are present. Therefore, although our tool detects that the source and the target products have different common methods, these transformations do not introduced behavioral changes. As we mentioned in Section III-A, the adapted version of SAFE REFACTOR does not consider refinement when the source and the target SPLs have different public methods, that is why the tool produces a false negative.

Confirming the developers expectations, our tools did not find behavioral changes in Subjects 9, 10, 13 and 14. These evolution scenarios modified FM, CK, and code, adding an optional feature. The optimized tools identified that the changes did not affect the original features. Therefore, they optimized the checking by only performing Steps 1 and 2.

Subjects 11 and 12 present behavioral changes illustrated in Section II, where incorrect associations between assets and expressions in the CK resulted in behavioral changes in products of the SPL. The first tool identified these behavioral changes. TaRGeT developers confirmed that these behavioral changes were reported a few days after delivering the releases to the clients. This illustrates the challenges for SPL evolution due to the effort of synchronizing all artifacts and checking all product configurations. The behavioral changes could be revealed earlier using our tools. The IMPACTED CLASSES tool achieved significant reduction in the time for evaluating Subject 7. While ALL PRODUCT PAIRS and IMPACTED PRODUCTS spent 29 and 13 minutes to finish, IMPACTED CLASSES took just about 9 minutes.

D. Discussion

In our experiment, the optimized tools were the most efficient tools for evaluating transformations that only change the FM, due to their optimization for reasoning about FM and CK changes. Therefore, developers should use them in this scenario. Meanwhile, IMPACTED CLASSES was the best tool on transformations that only change code, leading to reductions of up to 70% in time (Subject 7). However, as we presented in Section III-C2, the other tools are safer, since IMPACTED CLASSES needs to make some assumptions to be performed. Besides, we observed that the cost to run IMPACTED CLASSES increases as the number of changed classes grows, getting closer to the cost of running ALL PRODUCT PAIRS. So, developers should analyze this trade-off between precision and time in each situation.

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 sittuations. In these cases, ALL PRODUCTS is more suitable to evaluate the transformation, though less faithful to the refinement theory.

For transfomations that only affect GUI elements, we observed in our experiments that none of the tools is able to detect non refinements, since SAFE REFACTOR is not able to identify behavioral changes in these scenarios.

Opdyke [15] 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. SAFE REFACTOR checks, but does not prove, 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, they have equivalent behavior. 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 Subjects 8 and 15. It may be better to evaluate the SPL with respect to its Facade considering only its public methods. We plan to adapt SAFE REFACTOR to evaluate the SPL 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 not common to include methods related to optional features in mandatory classes as we saw in Subjects 8 and 15, which leaded to false negative results of our tool.

E. Threats to Validity

Threats to internal validity are influences that can affect the measured execution time or the results of our tools. We use the same machine to test the subjects using the four tools to avoid influences in the measures.

Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. The pairs of source and target versions we got in TaRGeT's SVN were indicated by TaRGeT's development team as transformations where they only have the intention of making a safe evolution between two commits, and behavioral changes in transformations in Mobile Media were introduced by us. We cannot guarantee that those pairs are representative enough for transformations applied in other SPLs.

V. Related Work

Opdyke proposed refactorings as behavior-preserving program transformations [15]. Initially, the definition supported the iterative design of object-oriented application frameworks. In practice, they evaluate behavior preservation by successive compilation and tests. Although Opdyke's work and later refactoring definitions apply to frameworks, which current SPL development uses, in a SPL, we have a number of products instead of a single program.

Alves et al. [16] informally present a SPL refactoring definition, based on FM changes that maintain or increase configurability. They propose FM transformations that conform to this definition. Borba [11] initially propose the formalization of the SPL refinement notion. They also illustrate different kinds of refinement transformation templates that can be useful for deriving and evolving SPLs. According to their definition, in a 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.

Moreover, Borba et al. [5] propose an extended version of this previous formalization, where they explore properties that justify stepwise and compositional evolution of SPL artifacts. The formalization and its properties are also proven sound with a theorem prover. We base the optimizations used in our tools for checking refinements on their work.

Thüm et al. [17] classify evolution of a FM in four categories: refactorings for changes that do not add or remove products; generalizations for changes that add new products without remove the existing ones; specializations for changes that remove products without add new products; finally, arbitrary edits for other cases. The notion of refinement we use 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. [18] 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 [5].

Thaker et al. present techniques for verifying type safety properties of AHEAD [19] SPLs using FMs and SAT solvers [20]. 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 [21] 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 the notion of SPL refinement that we used.

A number of approaches [22], [23], [24], [25] focus on refactoring a product into a SPL, not exploring SPL evolution in general, as we do here. Kolb et al. [22] discuss a case study in refactoring legacy code components into an SPL implementation. They define a systematic process for refactoring products with the aim of obtaining SPLs assets. There is no discussion about FMs and CK. However, like we do in this work, they check behavior preservation and configurability of the resulting SPLs by testing. Similarly, Kastner et al. [25] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs. As discussed here and elsewhere [11], [5] these are not adequate for justifying SPL refinement and refactoring. Trujillo et al. [23] go beyond code assets, but do not explicitly consider transformations to FM and CK. They also do not consider behavior preservation; they indeed use the term "refinement", but in the different sense of overriding or adding extra behavior to assets.

Liu et al. [24] 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. [26] 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 SAFE REFACTOR. 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. [27] propose a 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 [8]. We believe this tool is complementary to our approaches, since we could check if FLiP transformations are SPL refinements indeed, according to the definition we use [11], [5], [6].

VI. CONCLUSIONS

In this paper, 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. [5]. The suitability of each tool depends on the kind of change and on user's constraints regarding time and reliability. We evaluate them in evolution scenarios applied by developers to real SPLs ranging up to 32 KLOC and 1512 products. We found that two out of the ten real evolution scenarios introduced behavioral changes. This illustrates the challenges for SPL evolution due to the effort of synchronizing all artifacts and checking all product configurations. We aim at making SPL evolution safer.

As future work, we plan to evaluate our tools in more real case studies. Besides, we can improve performance of our tools by using incremental compilation, and parallelism to reduce the needed time for evaluation. We also plan to develop an Eclipse plugin version of them.

ACKNOWLEDGMENT

We would like to thank CNPq and CAPES/PROCAD, Brazilian research funding agencies, and National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work. In addition, we thank SPG[3] members for feedback and fruitful discussions about this paper.

[3]http://twiki.cin.ufpe.br/twiki/bin/view/SPG/WebHome

REFERENCES

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

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

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

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

[5] P. Borba, L. Teixeira, and R. Gheyi, "A theory of software product line refinement," in *ICTAC*, 2010, pp. 15–43.

[6] ——, "A theory of software product line refinement," *Theoretical Computer Science*, 2012.

[7] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba, "Investigating the safe evolution of software product lines," in *GPCE*, 2011, pp. 33–42.

[8] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Software*, vol. 27, pp. 52–57, 2010.

[9] F. Ferreira, L. Neves, M. Silva, and P. Borba, "Target: a model based product line testing tool," in *Tools Session at CBSoft*, 2010.

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

[11] P. Borba, "An introduction to software product line refactoring," in *GTTSE III*, ser. LNCS, vol. 6491, 2011, pp. 1–26.

[12] R. Gheyi, T. Massoni, and P. Borba, "A theory for feature models in alloy," in *1st Alloy Workshop*, 2006, pp. 71–80.

[13] L. Teixeira, P. Borba, and R. Gheyi, "Safe composition of configuration knowledge-based software product lines," in *SBES*, 2011, pp. 263–272.

[14] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2012.

[15] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, UIUC, 1992.

[16] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring product lines," in *GPCE*, 2006, pp. 201–210.

[17] T. Thum, D. Batory, and C. Kastner, "Reasoning about edits to feature models," in *ICSE*, 2009, pp. 254–264.

[18] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[19] D. S. Batory, "Feature-oriented programming and the AHEAD tool suite," in *ICSE*. IEEE Computer Society, 2004, pp. 702–703.

[20] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *GPCE*, 2007, pp. 95–104.

[21] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek, "Refactoring product line architectures," in *1st International Workshop on Refactoring: Achievements, Challenges, and Effects*, 2003, pp. 23–26.

[22] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line," in *ICSM*, 2005, pp. 369–378.

[23] S. Trujillo, D. Batory, and O. Diaz, "Feature refactoring a multi-representation program into a product line," in *GPCE*, 2006, pp. 191–200.

[24] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE*, 2006, pp. 112–121.

[25] C. Kastner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *SPLC*, 2007, pp. 223–232.

[26] C. H. P. Kim, D. S. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *AOSD*, 2011, pp. 57–68.

[27] S. Soares, F. Calheiros, V. Nepomuceno, A. Menezes, P. Borba, and V. Alves, "Supporting software product lines development: Flip - product line derivation tool," in *In OOPSLA Companion*, 2008, pp. 737–738.