Patching Product Line Programs*

Martin Kuhlemann
Faculty of Computer Science
University of Magdeburg, Germany
martin.kuhlemann@ovgu.de

Martin Sturm
Faculty of Computer Science
University of Magdeburg, Germany
MartinSturm@gmx.net

ABSTRACT

Software product line engineering is one approach to implement sets of related programs efficiently. Software product lines (SPLs) can be implemented using code transformations which are combined in order to generate a program. A code transformation may add functionality to a program or may alter its structure. Though implemented with less effort, a single malfunctioning SPL program is harder to patch because patches must effect the SPL transformations which the program was generated from. In this paper, we present a new approach to patch programs of a transformation-based SPL. We demonstrate the feasibility of this approach using a prototype.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms, Design

1. INTRODUCTION

A *software product line (SPL)* is a set of related programs which are generated from a shared code base [6]. SPL programs are defined using *features* (user-visible program characteristics [17]) and programs of an SPL differ in features. Features can be implemented by program transformations, which add functionality to a program or alter the structure of a program. An SPL program then is generated by selecting features and executing code transformations which implement those features. Reusing transformations across SPL programs reduces the overall effort to implement the SPL programs. In prior studies on transformation-based SPLs, however, we and others observed that errors were hard to track and remove [2, 20, 33]. The solutions presented accordingly concentrate on syntactic correctness but semantic correctness is not discussed, i.e., patching SPL programs is not discussed.

As a first contribution of this paper, we provide an analysis on how SPL programs can be patched (manually edited). In particular, we compare the *patching of SPL transformations* with the *patching of generated programs*. As sample transformations used in SPLs, we concentrate on superimpositions and refactorings. From our analysis we argue that patching the *transformations* of an SPL can be inappropriate – we need techniques to also patch the *generated* code.

As a second contribution, we present and demonstrate a new approach for propagating patches from generated SPL programs to SPL transformations. That is, after the developer stepped through and patched the generated program, a propagation tool identifies and propagates the patches to the SPL transformations. In this approach, we combine an index data structure with a transformation history data structure to optimize patch propagation. We demonstrate the feasibility of this approach with our prototype.

*This paper summarizes and extends the Master's Thesis of Martin Sturm [31]. An extended version of this paper with more technical details has been published before as a technical report [21]. The authors thank Christian Kästner, Don Batory, and Marko Rosenmüller for helpful comments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FOSD'10 October 10, 2010, Eindhoven, The Netherlands
Copyright 2010 ACM 978-1-4503-0208-1/10/10 ...$10.00.

2. TRANSFORMATIONS IN SPLS

Superimposition. A number of approaches use superimposition transformations to implement SPLs, e.g., [2, 4, 25]. A superimposition creates classes in its input program and executes class refinements on this input program. A *class refinement* creates members in input-program classes and executes method refinements. *Method refinements* create statements in input-program methods. We use feature-oriented programming to represent SPL techniques that use superimposition transformations; further, we use Jak as a sample feature-oriented language [4]. Jak extends Java by superimposition mechanisms.

In Figure 1a, we sample code which is similar to code of an SPL from prior work where we integrated programs of a compression-library SPL with environments, i.e., we bridged incompatible structures between them [19]. A superimposition transformation *Base* is defined to transform a possibly empty input program by creating a class ZipArchive, i.e., *Base* encapsulates ZipArchive. Superimposition *Stats* encapsulates a class refinement of ZipArchive which extends ZipArchive in the input program of *Stats*. This class refinement encapsulates members (Lines 4-7) and a method refinement (Lines 8-10). The method refinement ZipArchive.getID

Transformation *Base*

```
1 class ZipArchive{
2   int getID(){return 1;} }
```

Transformation *Stats*

```
3 refines class ZipArchive{
4   int counter=0;
5   void count(){
6     counter = counter/1;
7     System.out.print(counter); }
8   int getID(){
9     count();
10    return Super.getID(); } }
```

Transformation *MakeCompatible*

```
11 Rename method: ZipArchive.getID() ↦id
```

(a) SPL transformations.

```
1 class ZipArchive{
2   int counter=0;
3   void count(){
4     counter = counter/1;
5     System.out.print(counter);}
6   int getID(){
7     count();
8     return 1;} }
```

(b) SPL product generated from *Stats*, *Base*.

```
1 class ZipArchive{
2   int counter=0;
3   void count(){
4     counter = counter/1;
5     System.out.print(counter);}
6   int id(){
7     count();
8     return 1;} }
```

(c) SPL product generated from *MakeCompatible*, *Stats*, *Base*.

Figure 1: Superimpositions and their composition result.

of *Stats* extends method ZipArchive.getID in the input program of *Stats* (method created by *Base*) and creates statements in this method. The extended method is called with Super (Line 10). The result of executing *Base* and *Stats* from Figure 1a is shown in Figure 1b. A class ZipArchive there encapsulates the members of both ZipArchive class fragments it was generated from (members getID, counter, count). Method getID encapsulates the code generated from getID of *Base* and of *Stats*.

Refactoring. Program generation may involve configuring the program's structure using selectable refactorings. *Refactorings* are transformations which alter the structure of a program but do not alter its functionality [13]. For example, renaming a method of a program and updating all method calls is a Rename-Method refactoring [13]. We use *refactoring feature modules (RFMs)* [19] to represent SPL techniques which use refactoring transformations.

We introduced RFMs to allow configuring the structure of SPL programs [19]. RFMs allowed us to reuse SPL programs in different incompatible environments in which they could not have been reused as is before. One selection of refactorings then allowed us to reuse SPL programs in one environment and another selection allowed us to reuse *the same programs* in another environment.

In Figure 1a, we defined an RFM *MakeCompatible* in the SPL code base. Executing *Base*, *Stats* and *MakeCompatible* generates a class ZipArchive with members id, count, and counter but no member getID, see Figure 1c.

Refactoring transformations have preconditions that specify which properties a piece of code must fulfill such that the executed transformation does not alter the piece's functionality [28]. The above refactoring of *MakeCompatible* (Rename Method: ZipArchive.getID $\mapsto$ id) requires that method ZipArchive.getID exists and that no method ZipArchive.id exists in the code to refactor. (Further preconditions exist, but are not important for now.)

Refactorings can replace pieces of code one-by-one, can merge code, can multiplex code, and can create code in their respective input programs. Since transformations with these abilities are considered non-trivial [1, 8, 12, 38], refactorings are non-trivial transformations in SPL technology. For example, a Rename-Class refactoring inside an RFM replaces (renames) a class of an input program one-by-one by a class in the according output program [13]. A Pull-Up-Method refactoring inside an RFM merges multiple methods of an input program into one method in the according output program.[1] A Push-Down-Field refactoring inside an RFM multiplexes a single field of an input program in the according output program.[2] An Encapsulate-Field refactoring inside an RFM creates a get and a set method for a field and no code is removed for that [13].

3. PROBLEM STATEMENT

A generated SPL program may work incorrectly due to bugs. As an example, we placed a bug in the code of Figure 1. The program in Figure 1c should print the operation count (consecutive numbers) but it prints zeros instead. The reason is that the code in Line 4 is incorrect and should be patched to "counter=counter+1;". Finally, however, this patch must affect Line 6 of the SPL transformations (Fig. 1a). To patch SPL programs, we can (a) patch the *transformations* (Fig. 1a) and then regenerate the program, or (b) patch the *generated* program (Fig. 1c) and later possibly propagate the patches to the transformations. We now discuss strengths and weaknesses of both approaches.

3.1 Mapping Problem

SPL transformations like superimpositions and refactorings are executed one after the other such that the overall mapping of code in the generated program to code in transformations is complex [12, 36]. A piece of code in the generated program might be the result of merging, multiplexing, *and* replacing code during program generation, so we should hide this complex mapping from the developer. Additionally, the code shown to the developer should be in a language which she is familiar with.

When patching generated programs, a patch must be propagated to an SPL transformation T (we call this transformation *target*). For that, all SPL transformations $\mathbb{T}$ that follow T must be inverted (undone) in the generated program. To invert a refinement we can instantiate a remove operation; for a refactoring A, we can instantiate a refactoring of which the type is hard-coded to invert A. Patches may prevent a transformation of $\mathbb{T}$ being inverted. For example, adding a method ZipArchive.getID to the code of Figure 1c prevents inverting the transformation *MakeCompatible* (Re-

[1]Pull-Up Method moves a subclass method to a superclass and removes equivalent methods of other subclasses [13].

[2]Push-Down Field generates a copy of the pushed field in multiple subclasses and removes the superclass field [13].

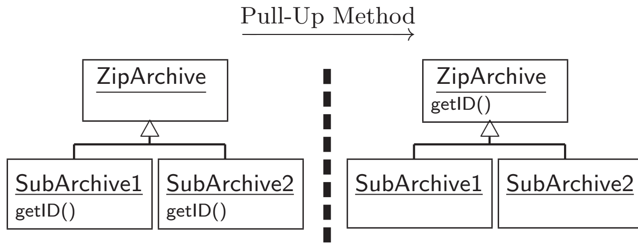

Figure 2: Pull-Up Method refactoring merges code.

name Method: ZipArchive.getID $\mapsto$ id) because otherwise ZipArchive.getID would be generated twice in SPL programs (which is an error in most languages). Approaches to patch generated programs, thus, need a *fallback strategy*. The code shown to the developer is in a language (Java) which is very similar to the language the developer used to implement the SPL (Jak+RFM), i.e., she should be familiar with it.

When patching SPL transformations and the code executed in the generated program was created by a refactoring (code did not exist before), there is no code on the level of the transformations to show to a developer accordingly [38]. Showing transformation definitions instead is not an option as they might not show the code they generate, either. For example, RFMs define properties of code to generate but RFMs do not include this code [19], e.g., they define which field to encapsulate but do not include get and set methods. However, skipping this code is error prone [12].

If an executed method was merged by a refactoring from multiple methods, each of these methods in the transformations is a valid (with respect to the generated program) mapping value from the executed, generated method [38]. Showing the wrong method with respect to the transformations, however, causes confusion. In the example of Figure 2, a Pull-Up-Method refactoring merges SubArchive1.getID and SubArchive2.getID to ZipArchive.getID. Assume, the patch tool defines SubArchive1.getID to represent the executed, merged method ZipArchive.getID on the level of the transformations. But if SubArchive2.getID was called on the level of transformations the developer is confused seeing SubArchive1.getID.

If executed code was merged by a refactoring from multiple methods, then breakpoints set in the transformations to one of the original methods can match too often or too rarely [1]. For example, methods SubArchive1.getID and SubArchive2.getID in Figure 2 are merged by a Pull-Up-Method refactoring. Assume, the patch tool maps the executed, merged method ZipArchive.getID to SubArchive1.getID which hosts a breakpoint in the transformations, then this breakpoint will match for SubArchive2.getID, too incorrectly. A breakpoint set to SubArchive2.getID (not referenced from the generated program) will never match. This nondeterminism hampers stepping through the program.

If executed code was generated by a refactoring which multiplexes code, multiple values of variables from the generated program might need to be merged on the level of transformations – this, however, might be impossible. As an example, consider a Push-Down-Field refactoring on a static field which generates multiple static fields in the generated program; these generated fields can expose different values in the executed program but can only expose one on the level of transformations. According values must be merged to be presented as one value for the single (pushed) field in the transformations. However, this is not possible generally. Variable values then cannot be analyzed.

The code shown to the developer is in the language the developer used to write the SPL, i.e., she should be familiar with it. (Solutions exist for problems similar to the problems above [1, 14, 38]. However, approaches are limited to languages for which conditional breakpoints and path analyses can be compiled into binaries.)

3.2 Scattering Program Code

During patching an SPL program a developer should concentrate on this program's bug in the first place. Just in the second place other SPL programs should be considered.

When patching generated programs, code of other SPL programs, i.e., code that does not contribute to the patched program, is hidden and errors in these other programs are postponed until finishing program patching. Nevertheless, the patches can be checked automatically against all SPL programs, e.g., during propagation, and an error can be reported when a patch introduces a program in error [33].

When patching transformations, code that contributes to one SPL program is scattered across transformations. For that, the developer must execute the transformations *in-mind* in order to foresee the code actually executed when stepping over it, e.g., when stepping over a method call. The developer needs similar knowledge to decide which methods she can call in a patch of a transformation ("How do input programs of a transformation-to-patch look like?"). Note, if tools would visualize generated code they switch to the patch-generated-programs approach.

As an example of in-mind transformation execution, reconsider Figure 1a. In these SPL transformations, a developer may want to add to superimposition *Base* a method that calls Element.id (not depicted). As Element.id is undefined in *Base* the developer must generate the patched program in-mind to verify that she is able to call this method in *Base*. Transformations in the SPL code base which do not contribute to the patched program distract the developer from the bug to repair.

Code in unpatched superimpositions might be replaced or overridden by a patch, accidentally, in the currently patched SPL program or others. For example, when a developer applies a patch (not depicted) in Figure 1a such that she adds a new superimposition *AfterStats* which follows *Stats* and such that she adds a method ZipArchive.count to *AfterStats*, then this *AfterStats* method replaces ZipArchive.count of *Stats*. It can get worse. If *Stats* would not contribute to the patched program but to a different program, the patch could replace count in this other program, unnoticed. Current approaches which validate the whole SPL [33] may help but executing them during patching hampers implementing the patch. When a method created in a patch *overrides* a method in a different SPL program accidentally current mechanisms do not help; especially, when the developer intended to override some different method.

3.3 Bounded Quantification Problem

Bounded quantification is a guideline to reduce complexity in transformation systems [23, 26]. Bounded quantification restricts code generated by a transformation to only access code which exists in the transformation's input program, i.e., which has been generated by preceding transformations.

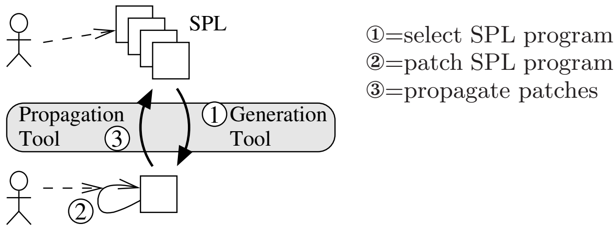

Figure 3: Use case for patching SPL programs.

Index Key	Index Value
ZipArchive.id	[*Base::*ZipArchive.getID, *Stats::*ZipArchive.getID]
ZipArchive	[*Base::*ZipArchive, *Stats::*ZipArchive]
ZipArchive.counter	[*Base::*ZipArchive.counter]
ZipArchive.count	[*Stats::*ZipArchive.count]

Table 1: Sample index for code of Figure 1c.

Bounded quantification should hold before patching and after, e.g., no developer/tool should patch a class in a transformation A to subtype a class which is generated by a follower transformation of A.[3] Developers should be advised in which transformation and how to implement a patch.

When patching generated programs, a patch is detected and propagated automatically. Thereby, the tool can advise in choosing a transformation to host the patch (target). The tool can further check that bounded quantification is not broken with this advice, even in other SPL programs.

When patching transformations, the developer may realize during patching that with the patch just implemented she breaks bounded quantification. Then she might have to move the patch (manually) into another transformation. Additional complexity is put upon the developer when patches break *subsequent* transformations (maybe in other SPL programs). In the example of Figure 1a, a developer, who aims to patch superimposition *Base* and, for that, adds a method ZipArchive.id, breaks the follower RFM *MakeCompatible* because this RFM requires ZipArchive.id not to exist in its input program. With a growing number of superimposition and refactoring transformations, the restrictions imposed by transformations, which follow a transformation to patch, become opaque and unmanageable [12].

3.4 Summary

Patching SPL transformations may present incorrect or no code to the developer (cf. Sec. 3.1), may show scattered code (cf. Sec 3.2), and may require to re-implement patches (cf. Sec. 3.3). Patching generated programs may "just" require a fallback strategy when transformations cannot be inverted.

4. PATCHING THE TRANSFORMATIONS

Some researchers argue to step through code and to patch it on the level of transformations (there: high-level code), e.g., [7, 15, 37]. From our analysis and in line with others [1, 8, 12, 36] we argue that stepping through code can be meaningful at every level of abstraction. We show now how code generated from transformations can be patched, too.

4.1 Conceptual Process

We propose to let the developer step through and patch the generated program, and to propagate *automatically* each patch to the best SPL transformation; this use case is depicted abstractly in Figure 3. The propagation tool is proposed to *find and link patches*, *prepare the propagation*, *perform the propagation*, and to *save the propagation*.

[3]Some languages used for SPLs, e.g., Jak or AspectJ, do not enforce bounded quantification. Patches to SPLs written in these languages can break bounded quantification.

Find and link patches. At first, the propagation tool should compare the patched program with an unpatched version of this program to find patches. To ease propagation later, the tool should link each patch to a qname (abbreviation for fully-qualified name) which encapsulates the patch. In our example of Figure 1, the propagation tool should find the patch in Line 4 of Figure 1c and should link it to qname ZipArchive.count.

Prepare propagation. To advise *where*, i.e., to which SPL transformation, to propagate a patch best later (we call such transformation *target*), the propagation tool should calculate the origins of generated members and classes. For that, it should analyze the executed superimposition transformations and refactoring transformations in execution order. The propagation tool should record in an *index structure* for every generated qname a (list of) qname from the transformations of which the code includes the generated code. If there is no such qname in the transformations, i.e., when an RFM creates code, then an empty list is recorded as index value for the qname of this code.

An index for the SPL program of Figure 1c is given in Table 1. The index key is a qname from the generated program and the index value is a list of qnames from code of the transformations, prefixed with these transformations. For example, ZipArchive.id is indexed to be generated from ZipArchive.getID in transformations *Base* and *Stats*.

To advise *how* a patch should be implemented in the target later, the propagation tool should record all code transformations which executed in a *transformation history*. In the transformation history of Figure 1c, *MakeCompatible* is recorded to affect *Base* and *Stats*.

Perform propagation. To perform patch propagation, the propagation tool first should calculate a good target and then invert all transformations on the patch which executed after the target during program generation. The tool should calculate a good target in three steps: First, the tool analyzes the qnames which the patch relates/references to, e.g., qnames of called method and hosting classes – the origin of these qnames is the best target for hosting the patch. Second, to avoid that an inverted transformation cannot be reperformed after propagation, the propagation tool should, for every transformation to invert, analyze the qnames and their relations in the respective transformation's input program. That is, the described index should also keep qnames deleted during program generation and tag them. If qname relations required by a transformation conflict a patch (disallow to invert a transformation), then we apply a *fallback strategy*. Third, the propagation tool should check whether bounded quantification is broken when the patch is propa-

gated to its target – if so, the propagation tool should adapt the target to be the *follower* of all transformations which introduce code the patch references (required condition for bounded quantification).

The fallback strategy we propose (there might be more): When inverting a transformation fails, the propagation tool creates a transformation which follows the non-invertible transformation; a transformation which will replace the erroneous code in future SPL programs.

We want to exemplify patch propagation with the priorly discussed patch for Figure 1c ("counter=counter/1;" $\mapsto$ "counter=counter+1;"). The patch is found and linked to ZipArchive.count. In the index shown in Table 1, ZipArchive.count is recorded to be generated in *Stats*, so *Stats* is calculated as target for the patch. The patch does not reference other qnames (bounded quantification cannot break) and so the target remains *Stats*. The propagation tool finally inverts *MakeCompatible* on the patch (nothing changes) and provides this propagated patch as advice to the developer. If *MakeCompatible* could not be inverted in the presence of patched ZipArchive.count, the tool should advise our fallback strategy, i.e., it should insert a new transformation as a follower of *MakeCompatible* with patched ZipArchive.count.

When a qname exists in the index but maps to an empty list of value qnames (the piece of code got created by a refactoring), then the tool should propagate the patch along the reverse global sequence of program transformations. Thereby, it should invert SPL transformations until one transformation identifies the patched piece of code as "self-generated" and provides a target. As an example, assume the patched ZipArchive.count would have been created by *MakeCompatible* (e.g., when *MakeCompatible* is an Encapsulate Field refactoring) – then ZipArchive.count would occur in the index but would map to an empty list. In this case, the transformations from *MakeCompatible* to *Base* would be inverted. The inversion process of *MakeCompatible* would stop this target-less propagation because at first it defines a new target for the patch. In this special case of *MakeCompatible* creating count, the target would be a new follower of *MakeCompatible* because RFM *MakeCompatible* does not encapsulate code to patch. If *MakeCompatible* would have merged ZipArchive.count from multiple pieces of code, then the inversion algorithm for *MakeCompatible* would generate multiple targets and propagation continues.

Save propagation. We use metrics (origins of referenced code) to identify the best target for a patch and so this target might be suboptimal semantically though meaningful and correct syntactically. A propagation tool, thus, should advise a mapping to the developer but should ask the developer to confirm. For instance, the tool could save the patched SPL transformations separately such that the developer can decide whether to accept the propagation. If she accepts, the original SPL transformations are replaced.

4.2 Unsupported Patches & Transformations

The transformations available for program generation limit the patches which can be propagated from the generated program to the transformations. Depending on the transformation language, no fallback strategy seems available.

As a first example, superimpositions in Jak cannot replace constructors generated by preceding transformations [29]. Patches in constructor bodies *must* be propagated to the transformation which created the constructor. If they cannot be propagated, we emphasize the advice from the Jak documentation to *extract constructors into initialization methods* [29]; methods which then can be replaced with our fallback strategy.[4]

As a second example, code removed from the generated program might not be allowed to be removed in the transformations, accordingly. If the deletion in the generated program prevents to invert a transformation, a propagation tool cannot propagate the patch but also cannot delete the qname by adding a superimposition (because superimpositions can only generate code) nor by adding a refactoring (because refactorings can only change code structure).

Refactorings (e.g., of RFMs), refinements (e.g., of Jak) as well as aspects [18] and rewrites [35] can be inverted such that patch propagation is possible, but this is not the general case. Higher-order rewrites (where pieces of code are matched by incomplete patterns) [34] can only be inverted when code matched with wildcards can be reconstructed. If a transformation cannot be inverted, we propose to apply our fallback strategy.

The mapping from the generated program towards the transformations is specific to tools. If different tools translate the same set of transformations differently, the index creation tool must be parameterized with the transformation tool used. For example, for Jak at least 2 different tools with 2 different translations into Java exist [3]. (Note, this situation adds effort in every solution approach.)

5. PROTOTYPE & DEMONSTRATION

We demonstrate that stepping through and patching the generated program and that propagating patches afterwards *is feasible* for superimposition- and refactoring-based SPLs. That is, we implemented the above concepts prototypically and used the tool in a demonstrating example.

Our prototype finds patches and links them to qnames (Phase 2 in Sec. 4.1), calculates the best target for every patch (Phase 3), and stores the propagation advice separately (Phase 4). To do this, for every superimposition (analyzed in composition order), the prototype collects the qnames of code which is generated in these superimpositions as index keys and as index values. Recorded transformations are executed on index keys but not on index values. Thus, the index in the end maps qnames of the generated program to qnames of the transformations. Our prototype currently supports one transformation tool for Jak and one for refactoring feature modules (RFMs) – error detection based on qname *relations* during program generation is not yet implemented (keeping and tagging deleted qnames). For more implementation details, simplifications, and for more demonstrating examples please consult [21, 31] – they are omitted for readability.

Demonstrating example. We demonstrate the propagation approach using the *Graph Product Line (GPL)* which has been proposed to be a standard benchmark for SPL technology [22]. Specifically, we use a version from prior work in which we added RFMs to GPL in order to integrate GPL

[4] For patched field initializations (fields also cannot be replaced in Jak superimpositions) we envision to encapsulate their initialization in methods, too, which can be replaced.

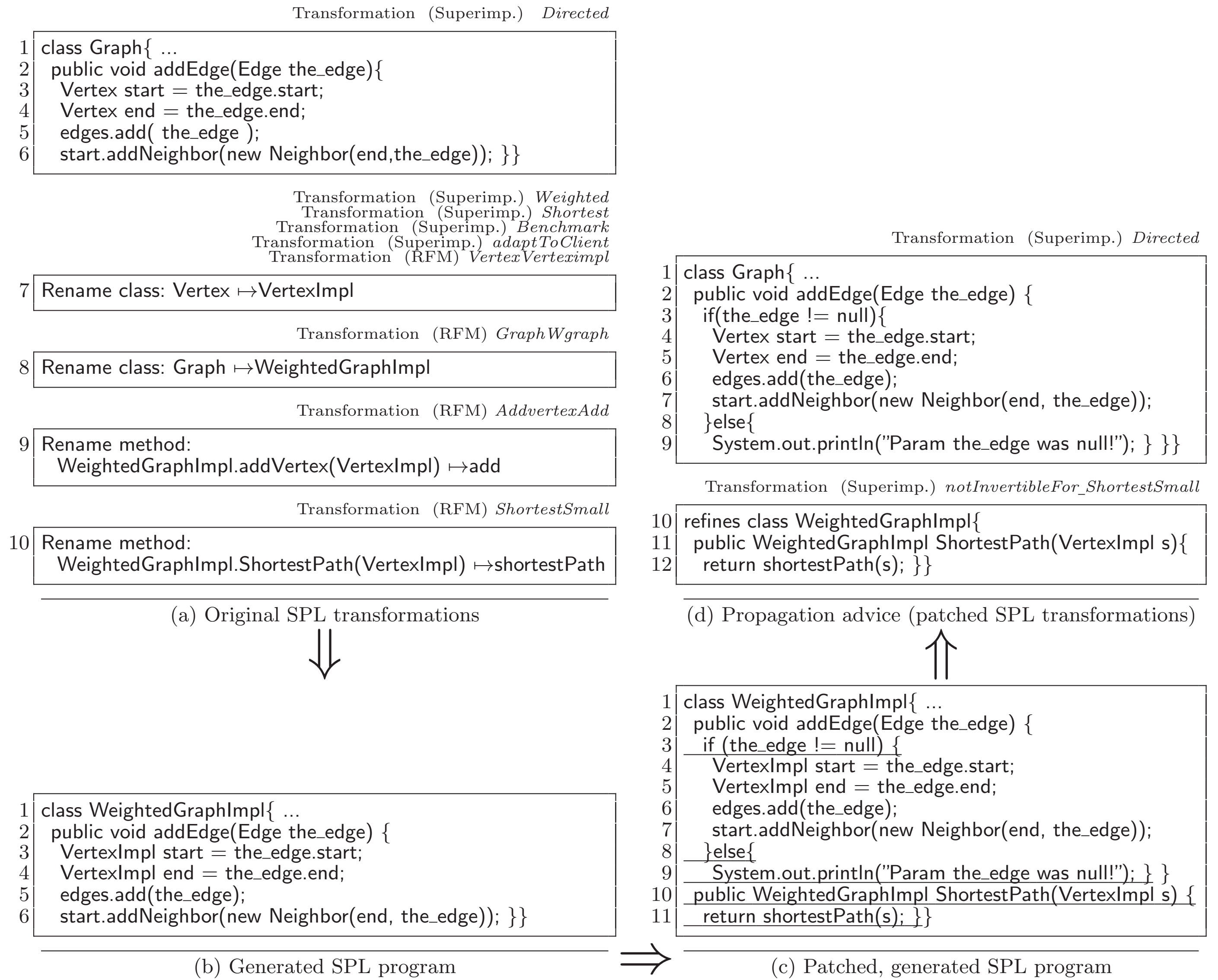

Figure 4: Stepping through and patching the code of the GPL class Graph/WeightedGraphImpl.

programs with incompatible environments [19].[5] The refactorings we apply, Rename-Class and Rename-Method, pose an important fraction when integrating programs [19].

We selected 9 features from GPL which correspond to program transformations which in turn generate a compilable program; 5 superimpositions and 4 RFMs. In Figure 4a, we list the transformation names in execution order (top-down) and show relevant code snippets from these transformations. The superimpositions generate and refine the classes Edge, Graph, Neighbor, and Vertex. The RFMs rename class Vertex into VertexImpl, class Graph into WeightedGraphImpl, method WeightedGraphImpl.addVertex(VertexImpl) into add, and method WeightedGraphImpl.ShortestPath(VertexImpl) into shortestPath. With these RFMs, the GPL program of the 4 superimpositions can be configured to be reusable as a library in a program with which this GPL program was incompatible before [19]. As we did not find bugs in the GPL program, we patched this generated program at will to cover interesting cases.

Using Figure 4, we want to demonstrate the proposed process in a practical scenario for patching class Graph. At first the developer executes the SPL transformations (Fig. 4a) to generate the SPL program (Fig. 4b). Later, this program is observed to not work properly, is stepped through, and is patched (Fig. 4c). By comparing the generated program (Fig. 4b) and the generated, patched code (Fig. 4c), the propagation tool finds three patches applied to class WeightedGraphImpl – we underline them in Figure 4c. The tool links these patches to qnames, e.g., it links the patch in Line 3 to WeightedGraphImpl.addEdge(Edge). The method addEdge(Edge) is found to be patched (not created) because an index key exists. As a result, the tool computes from its index that addEdge was generated lastly in the superimposition *Directed* which becomes target (cf. Fig. 4a). The propagation tool detects that code in the patch does not reference qnames others than the hosting method addEdge did

[5] We pruned the GPL version according to the current limitations of our prototype.

```
1  class VertexImpl{ ...
2   private boolean displayed = false;
3   public void display() {
4    System.out.print("Pred " + predecessor + " DWeight " +
       dweight + " ");
5    display$$eval$outWeighted$GG();
6    this.displayed = true;}
7   public boolean wasDisplayed(){
8    return displayed; }
9   public VertexImpl assignName(String name) {
10   this.name = name;
11   if(this.wasDisplayed()){
12     System.out.println("was already displayed!"); }
13   return (VertexImpl)this; }}
```

Figure 5: Patches to Vertex/VertexImpl.

before, so *Directed* remains target. The propagation tool inverts the 4 RFMs, which executed after target *Directed*, on the patch and advises the developer to replace method addEdge in transformation *Directed* (Fig. 4d, Lines 2-9). The second patch (Fig. 4c, Lines 8-9) is linked to the same qname WeightedGraphImpl.addEdge and is propagated together with the first one.

The third patch (Fig. 4c, Lines 10-11) requires the fallback strategy. The patch concerns a method ShortestPath, which was created during patching (no index key exists). The tool analyzes that ShortestPath solely references shortestPath (generated in *Shortest*) and thus the tool uses *Shortest* as target for ShortestPath. Next, the target *Shortest* is validated whether all RFMs that executed after *Shortest* can be inverted with ShortestPath.[6] The propagation tool cannot invert RFM *ShortestSmall* (Rename Method: WeightedGraphImpl.ShortestPath(VertexImpl) $\mapsto$ shortestPath) because this would make SPL transformations, which execute before *ShortestSmall*, generate two methods WeightedGraphImpl.ShortestPath(VertexImpl) in products – this is an error in most product languages. Following our fallback strategy, the propagation tool adds a superimposition *notInvertibleFor_ShortestSmall* as a follower of *ShortestSmall* (cf. Fig. 4d), a superimposition which then adds the patch ShortestPath to future programs. Note in Figure 4d, as *ShortestSmall* is not inverted, the tool advises to refine class WeightedGraphImpl (exists after *ShortestSmall*) instead of Graph.

In a final analyzed case, we added references towards qnames in a patch. In Figure 5, we underline the patches applied to class VertexImpl.[7] The field displayed and the method wasDisplayed got added, and methods display and assignName got patched to access the added field and method. Our propagation tool detects those accesses and for that advises to use superimposition *Shortest* (refines display lastly) as target for displayed. As wasDisplayed got added and solely accesses displayed, the propagation tool advises to propagate wasDisplayed to *Shortest* (target of displayed), too. If we would patch display and assignName in *Directed* (creates display and assignName), the reference to a *Shortest* method or field would break bounded quantification. For that, the patches to methods assignName and display are advised to be propagated to *Shortest*, too. In *Shortest* they replace the methods created in *Directed*. Summarizing, our tool advised well *where and how* to propagate patches which we made to an SPL program.

[6]Superimpositions are invertible in Jak naturally due to the Jak composer implementation. In other languages local variables might need to be transformed into fields.

[7]Line 5 of Fig. 5 shows a possible translation for Super.display(); inlining this call would remove the statement.

6. RELATED WORK

There is much work on how to relate generated and transformation code, e.g., for stepping through code [12, 27, 30, 32, 34, 35, 37]. In addition to this work, we *propagate* patches from generated SPL programs to the code base of an SPL. There is further work on how to propagate patches from generated programs to the code of a superimposition-based SPL [3]. In addition to this work, we support SPLs which are implemented by superimpositions *and refactorings*.

Compilers execute refactoring-like transformations (optimizations) on code which keep functionality. Patching these (one-of-a-kind) programs poses similar problems as we faced for SPLs [1, 7, 14, 15, 36, 38]. In contrast to according work, the transformations we considered (superimpositions and refactorings) cannot only change code structure but also add functionality. As a result, executed code exists in the generated program but has no origin in the (possibly empty) initial input program.

MolhadoRef inverts refactorings to reduce human interaction when integrating a patched program with a former revision of this program [10, 11]. Lynagh provides ideas similar to MolhadoRef for edits [24]. While both approaches propagate patches of a program toward a single program (an earlier revision), we propagate patches of a program toward the code base of a transformation-based SPL.

Bidirectional transformations (a.k.a. lenses) synchronize multiple related representations of elements where patches can be propagated in any representation [9, 16]. In the patch propagation problem we focused on, edits to the generated program may prevent the execution of inverse RFMs (we, thus, discussed a fallback strategy) – such situation may not occur for bidirectional transformations.

Design maintenance systems execute transformations in order to generate a program [5]. In design maintenance systems, maintenance deltas are transformations which are added to a transformation history during maintenance of the generated program [5]. Baxter indicates patch propagation toward an old abstraction (specification) [5], but this old abstraction is no code base of an SPL (instrumenting superimpositions and refactorings). Baxter did not explore backward integration mechanisms though.

7. CONCLUSIONS

In this paper we discussed a number of problems which occur when stepping through and patching a program generated from a transformation-based software product line (SPL). Specifically, we discussed problems of complex mappings of code between the transformations and the generated program, problems of scattered SPL program code, and problems of patches that increase complexity. We found that for SPLs implemented with transformations of superimpositions and refactorings, the *generated* code is a beneficial option for stepping and patching. We automated the propagation of patches from the generated program to the SPL transformations and demonstrated its feasibility.

For our approach, we combined index techniques and transformation histories (both known from other contexts) to aid patching in transformation-based SPLs. Our propagation tool links detected patches to fully-qualified names of the generated program. It calculates the best SPL transformation to host a patch. The tool finally advises *how* to integrate the patch with the SPL transformations.

8. REFERENCES

[1] A.-R. Adl-Tabatabai. *Source-level debugging of globally optimized code*. PhD thesis, Carnegie Mellon University Pittsburgh, 1996.

[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering – An International Journal*, 17(3):251–300, 2010.

[3] D. Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *GTTSE*, pages 3–35, 2006.

[4] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *TSE*, 30(6):355–371, 2004.

[5] I.D. Baxter. *Transformational maintenance by reuse of design histories*. PhD thesis, University of California at Irvine, 1990.

[6] P. Clements and L. Northrop. *Software product lines : Practices and patterns*. Addison-Wesley, 2006.

[7] D.L. Curreri, A.K. Iyengar, R.A. Biesele, and M.A. Ruscetta. Debugging optimized code using data change points, 2000. US patent #6,091,896.

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

[9] K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, pages 260–283, 2009.

[10] D. Dig. *Automated upgrading of component-based applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[11] D. Dig, K. Manzoor, R.E. Johnson, and T.N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *TSE*, 34(3):321–335, 2008.

[12] R.E. Faith. *Debugging programs after structure-changing transformation*. PhD thesis, University of North Carolina at Chapel Hill, 1998.

[13] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] J. Hennessy. Symbolic debugging of optimized code. *TOPLAS*, 4(3):323–344, 1982.

[15] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27(7):32–43, 1992.

[16] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM*, pages 178–189, 2004.

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

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[19] M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *ICSR*, pages 106–115, 2009.

[20] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *GPCE*, pages 177–186, 2009.

[21] M. Kuhlemann and M.Sturm. Debugging product line programs. Technical Report 6, Faculty of Computer Science, University of Magdeburg, 2010.

[22] R.E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, pages 10–24, 2001.

[23] R.E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In *SPLAT*, 2005.

[24] I. Lynagh. An algebra of patches, 2006. http://-urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf.

[25] M. Odersky. *The Scala language specification (version 2.7)*, 2005.

[26] D.L. Parnas. Designing software for ease of extension and contraction. In *ICSE*, pages 264–277, 1978.

[27] Z. Porkoláb, J. Mihalicza, and Á. Sipos. Debugging C++ template metaprograms. In *GPCE*, pages 255–264, 2006.

[28] D.B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[29] Software Systems Generator Research Group. *The jampack composition tool*. AHEAD tool suite v2008.07.22, manual.

[30] B. Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 1991.

[31] M. Sturm. Debugging Generierter Software nach Anwendung von Refactorings. Master thesis, University of Magdeburg, Germany, 2010. http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisSturm.pdf.

[32] Sun Microsystems, Inc. *JSR-000045 Debugging support for other languages 1.0 FR*, 2003.

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

[34] A. van Deursen and T.B. Dinesh. Origin tracking for higher-order term rewriting systems. In *HOA*, pages 76–95, 1994.

[35] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5-6):523–545, 1993.

[36] H. Venturini, F. Riss, J.-C. Fernandez, and M. Santana. A fully-non-transparent approach to the code location problem. In *SCOPES*, pages 61–68, 2008.

[37] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *SP&E*, 38(10):1073–1103, 2008.

[38] P.T. Zellweger. An interactive high-level debugger for control-flow optimized programs (summary). In *Software Engineering Symposium on High-Level Debugging*, pages 159–171, 1983.