Technical report

Nr.: FIN-006-2010

Debugging Product Line Programs

Martin Kuhlemann und Martin Sturm

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-006-2010

Debugging Product Line Programs

Martin Kuhlemann und Martin Sturm

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Martin Kuhlemann
Postfach 4120
39016 Magdeburg
E-Mail: martin.kuhlemann@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 06.04.2010

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Debugging Product Line Programs*

Martin Kuhlemann[1] and Martin Sturm[2]

[1] University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de
[2] 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 by code transformations which are combined in order to generate a program. A code transformation may add functionality to a base program or may alter its structure. Though implemented with less effort, generated programs are harder to debug because debug changes must effect the SPL transformations which the program was built from. In this paper, we present a new approach to debug programs (of an SPL) generated by program transformations.

1 Introduction

A *software product line (SPL)* can be used to implement a set of related programs from a shared code base [22]. Programs of an SPL differ in *features*, which are user-visible program characteristics [21], and programs are defined using features. Features can be implemented by program transformations, which add functionality to a base program or alter the structure of a program. A program is generated from an SPL by selecting features and executing code transformations which implement those features. In prior studies on transformation-based SPL technology, however, we and others observed that errors were hard to track and remove [38,24].

In this paper, we discuss different approaches to debug SPL products. In particular, we compare the *debugging of generating code* (transformations) with the *debugging of generated code*. As sample transformations used in SPLs, we concentrate on superimposition and refactoring. From our analysis and inline with others we reason that debugging the *generating* code of an SPL can be inappropriate [6, p.326] [12, p.22]. We need techniques to debug the *generated* code. While techniques to debug generating code have been presented before, we show how to assist the user in propagating debug changes from the generated code to the SPL transformations.

2 Background

SPLs can be implemented using a set of program transformations that are executed to generate different programs. In this section, we review transformations

* This paper summarizes and extends the Master's Thesis of Martin Sturm [36].

Superimposition *Base*

```
1  class Queue{
2    int get(){return 1;}
3  }
```

Superimposition *F1*

```
4  refines class Queue{
5    int get(){
6      track();
7      return Super.get();
8    }
9    int actions=0;
10   void track(){actions++;}
11 }
```

(a) Generating code.

```
1  class Queue{
2    int get(){
3      track();
4      return 1;}
5    int actions=0;
6    void track(){actions++;}
7  }
```

(b) Generated code.

Fig. 1. Superimpositions and their composition result.

used in SPL technology: superimposition and code restructuring. Later on, we discuss superimposition and code restructuring transformations with respect to debugging.

2.1 Superimposition

A number of approaches use superimposition transformations and are used to implement SPLs [5,28,3,2]. Superimpositions generate classes and *class refinements* in a base program. A class refinement generates members in existing classes and applies *method refinements*. Method refinements generate statements in existing methods. For our upcoming discussions we use *feature-oriented programming (FOP)* in our examples to represent SPL techniques that use superimposition transformations; further, we use Jak as a sample FOP language [5].

In Figure 1a, a superimposition transformation *Base* is defined to transform an (empty) program by generating a class Queue (*Base* encapsulates Queue). Superimposition *F1* encapsulates a class refinement of Queue. This class refinement encapsulates members (Lines 9-10) and a method refinement (Lines 5-8). The method refinement Queue.get of *F1* extends method Queue.get of *Base* by overriding and generates statements in this method (overridden method called with Super, Line 7). The result of executing *Base* and *F1* from Figure 1a is shown in Figure 1b. The generated class Queue encapsulates the members of both Queue class fragments it was generated from. Method get encapsulates the code generated from get of *Base* and of *F1*.

2.2 Code Restructuring

Program generation may involve the restructuring of code by *refactoring*. Refactorings are transformations which alter the structure of a program but do not

alter its functionality [29]. For example, renaming a method of a program and updating all method calls is a Rename Method refactoring [15].

Refactoring Feature Modules (RFMs) is one implementation where refactorings are transformations in SPLs [23]. RFMs were introduced to integrate libraries generated from SPLs with incompatible applications. RFMs transform the generated SPL product to expose a different structure than the classes and class refinements inside the superimposition transformations of SPLs. If we add an RFM *R1* (Rename Method: Queue.get↦first) to the code base of Figure 1a then executing *Base*, *F1* and *R1* generates a class Queue with members first, track, and actions but no member get.

Refactoring transformations have preconditions [32] that specify which properties a piece of code must fulfill such that the executed transformation does not change the piece's functionality. The above Rename Method refactoring of *R1* (Rename Method: Queue.get↦first) requires that method Queue.get exists and that no method Queue.first exists in the code to refactor.[3] Preconditions are common for all kinds of transformation.

3 Debugging Problems

To debug a product of an SPL, we consider two basic approaches:

- Debugging the *generating* code and then generate the corrected product.
- Debugging the *generated* code and later possibly propagate the debug changes to the generating code.[4]

For both debugging approaches we identified problems which are related to the mapping of code across its representations and the mapping of breakpoints (mapping problem), to in-mind execution of transformations (interface problem), and to program complexity (bounded quantification problem).

3.1 Mapping-Problem

When a user debugs the *generated* code, the debug changes must be propagated to the generating code. When the user debugs the *generating* code, the display of the user must be adjusted in the generating code according to the executed statements in the generated code. That is, the mapping of code elements *from the generated code toward the generating code* must be performed for both approaches.

Transformations of an SPL can be classified using well-known types of mappings between the generated and the generating code elements [6, p.323ff]. Some mapping types hamper debugging [6, p.326], so we show that refactorings cover them all:

[3] There are additional preconditions for Rename Method refactoring. These preconditions are not important for now.

[4] Possibly a bug fix for a certain customer should be evaluated for some time before propagating it into the generating code.

- In a bijective mapping (1↦1), a single code element in the generated code is generated by transforming one code element in the generating code (superimpositions). For instance, a Rename Class refactoring inside an RFM maps (renames) one class in the generating code to one class of the generated code [15]. Further refactorings which establish a bijective mapping include Rename Method, Move Field, and Move Method.
- In a surjective mapping (n↦1), a single code element in the generated code is generated by transforming multiple code elements in the generating code. For example, a Pull-Up Method refactoring inside an RFM maps (transforms) multiple methods from the generating code to one method of the generated code.[5] Further refactorings which commonly establish a surjective mapping include Pull-Up Field, Inline Method, and Inline Temp.
- In an injective mapping (1↦n), multiple code elements in the generated code are generated by transforming a single code element in the generating code. For example, a Push-Down Method refactoring inside an RFM maps (transforms) a single method of the generating code to multiple methods of the generated code.[6] Further refactorings which establish an injective mapping include Push-Down Field, Extract Method, Extract Superclass, Extract Class, and Extract Interface.
- In a partial mapping (0↦1), a code element in the generated code is generated without transforming a code element in the generating code. Partial mappings may occur when RFMs generate code. For example, an Encapsulate Field refactoring inside an RFM generates a get and a set method for a field, but the field is not transformed into those methods [15]. As a result, the methods get and set have no code element in the generating code they are transformed from. Further refactorings which cause this type of mapping include Extract Interface, Hide Delegate, and Self Encapsulate Field.

Transformations of the mapping types (n↦1), (1↦n), and (0↦1) have been argued to complicate debugging [6, p.326] [1,40,12].[7] SPL transformations like superimpositions and refactorings are applied one after the other such that the mapping of one transformation blurs with the mappings of the transformations that follow [12, p.14]. As a result for example, the mapping for Queue.first in Figure 2 is blurred. Queue.first is generated by a Rename Method refactoring but does not directly map to one method in the generating code. Instead Queue.first was generated from a number of methods by an Inline Method RFM before. Summarizing, the complexity of the mapping grows rapidly with a growing number of complex SPL transformations.

[5] Pull-Up Method copies a single copy of equivalent methods from subclasses to a superclass and removes the remaining equivalent subclass methods [15].

[6] Push-Down Method generates a copy of the pushed method in multiple subclasses and removes the superclass method [15].

[7] We consider Extract Method to by 1↦n-typed. Others, interpreted Extract Method as the merging of duplicate statements across methods – interpreting Extract Method as n↦1 mapping [40][12, p.13]. They analogously consider our n↦1-typed refactorings, like Inline Method, to be 1↦n-typed [1, p.143]. Our notion is based on the number of named code elements which exist before and after a refactoring.

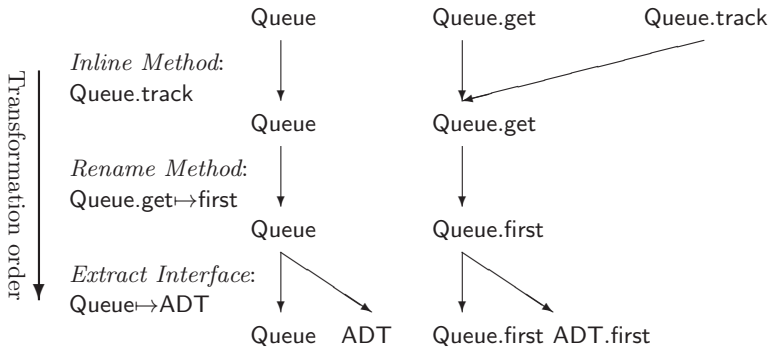


Fig. 2. Sample mapping between generating and generated code.

When debugging *generated* code the mapping problem may lead to situations in which the propagation of debug changes towards the generating code fails due to the debug changes (transformations may not be invertible due to the debug changes). According approaches, thus, must provide a fallback strategy.

When debugging *generating* code the mapping problem may lead to situations in which the code to display to a user is ambiguous or does not exist. For example, if the code executed in the product was generated by a partial-mapping-typed refactoring ($0 \mapsto 1$) there might be no generating code to display to a user for certain steps of the generated executed code. Displaying transformation descriptions instead is not an option as they might not show the code they generate, either, e.g., RFMs parameterize algorithms but do not include algorithms [23].

If the executed code was generated by a refactoring which establishes a surjective mapping ($n \mapsto 1$) there are multiple pieces of code in the generating code that are a valid mapping value from the generated executed code, i.e., which could be validly displayed to a user [40]. Displaying the wrong one, however, causes confusion.[8]

Furthermore, if code was generated by a refactoring which establishes a surjective mapping ($n \mapsto 1$) breakpoints set in the according generating code match too often or too seldomly. For example, breakpoints set to a method generated by a Pull-Up Method refactoring match more frequently than correct when they are set to a generating (pulled) element which got set as a mapping value for the generated code – the breakpoint will match for every pulled element in the generating code. A breakpoint set unknowingly to the wrong element in the generating code (not referenced from the generated code) will never match. This nondeterminism hampers debugging.

[8] In case of an applied Pull-Up Method refactoring we can display *one* of the equivalent methods in the generating code to represent the executed method which was generated out of them. But this choice might be incorrect according to the calling class in the generating code.

If the code executed in the product was generated by a refactoring which establishes an injective mapping (1↦n) multiple values of variables from the generated code might need to be merged – this, however, might be impossible. As an example, consider a Push-Down Field refactoring[9] on a static field which generates multiple static fields in the generated code; these generated fields can expose different values at debug time but cannot in the generating code. When debugging the generating code, according values must be merged to be displayed as one value for the single (pushed) field in the generating code. However, there is no general way, for example, to map the values of different Integer fields in the generated code to one Integer field in the generating code. As analyzing variable values is central to debugging imperative languages [39], being unable to map variable values is a serious problem when debugging imperative generating code.

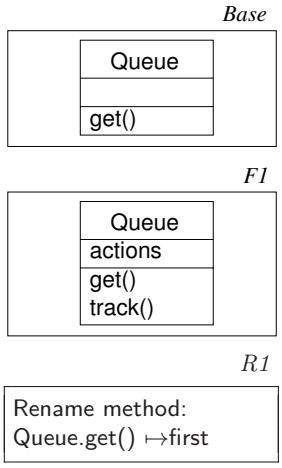


Fig. 3. Refactoring and superimposition-based SPL.

3.2 Interface Problem

Current development environments for SPLs which are based on superimposition (and refactoring) transformations require the user to execute the superimpositions in mind in order to understand the steps the debugger takes. That is, when debugging *generating* code the user must oversee which types and methods exist in the generated program, i.e., which methods can be called in a debug change. As a sample debug change in Figure 3, a user wants to add to superimposition *Base* a method that calls Queue.track. As Queue.track is undefined inside *Base* the user must generate the debugged program in mind to verify that she is able to call Queue.track. Transformations of an SPL which do not contribute to the debugged product further hamper in-mind product generation.

When debugging *generating* code, code elements in superimpositions might be replaced or overridden accidentally. For example, when a user applies a debug change to generating code and thereby adds a method, this new method may replace a method in the superimposition sequence of a different product. Current validation approaches alert when a method could be replaced in a class refinement for any product of the SPL [38]. Accidental method overriding, however, is only alerted when introduced by RFMs but not by a debugging user [24].

In the case of RFMs intermixing with superimpositions in their application sequence, the complexity of executing refactorings and refinements in mind increases further. For example, a call to a method get() in a superimposition may

[9] Push-Down Field moves a field of a superclass into those subclasses that use the field [15].

have to be written as top(int,String) because refactorings applied between the generation of get() and the method call.[10]

When debugging *generated* code, code of other SPL products is hidden and errors in other products are conjourned until finishing product debugging. The product does not have to be generated in mind.

We infer from the interface problem that tools to support debugging of an SPL product should hide SPL transformations which do not contribute to the debugged program. Nevertheless, the debug changes should be checked against all products of this SPL and an error should be reported when a debug change puts any product in error. Nevertheless, this error should not hamper the user in implementing a debug change. Error handling regarding other SPL products should be adjourned until finishing product debugging.

3.3 Bounded Quantification Problem

Bounded quantification allows code in (superimposition) transformations to only reference code of preceding transformations [27]. Bounded quantification reduces complexity in transformation systems, so it should hold before debugging and after. Debug changes are unforeseen and so they may break bounded quantification for the generating code. For example, adding members to classes, adding statements to methods, or changing signatures of members or classes are common debug changes but may break bounded quantification.[11] The user, thus, should be assisted where to put a debug change.

When debugging *generated* code the user can be assisted choosing which superimposition is best for hosting the performed debug change.

When debugging *generating* code, complexity of managing incomprehensible errors is put upon the user when debug changes break subsequent transformations. For example, in Figure 3, a user, who aims at debugging superimposition *Base* gets an error when she adds a method Queue.first because this method breaks the subsequent RFM *R1* (requires Queue.first not to exist). With an increasing number of superimposition and refactoring transformations, the restrictions imposed by transformations, which follow a superimposition to change, become opaque and unmanageable.

4 A Debugging Process for Transformation-based SPLs

We must be able to debug a generated SPL product using the *generating* and using the *generated* code [6, p.326] [12, p.22] [20]. The discussions above underline this issue. In this section, we present our approach to support the user in debugging the generated code and to assist her in propagating the debug changes to the generating code of the SPL.

[10] Add Parameter refactorings generate formal parameters in a method and require to pass actual parameters in calls [15]. Rename Method refactorings rename methods and update according calls [15].

[11] The user may reference code elements generated in a superimposition which applies later than the changed superimposition.

Fig. 4. A debugging process for SPLs built from complex transformations.

4.1 Conceptual Process

We summarize the process in Figure 4 and discuss its concepts below.

Debug generated code. We argue that users should be able to debug the *generated* code of their SPL product. This code is in a paradigm which superimposition and refactoring-based SPL techniques extend and which thus should be common to the user. When debugging the generated code, all tools that exist for off-the-shelf programs can be reused.

Find changes. After debugging the generated code we propose to compare the debugged program with the unchanged version of this program. By comparing the debugged code with the unchanged code, we can now identify the debug changes. For this task we can reuse off-the-shelf tools like *diff*[12].

Compute origins. We propose to calculate the origin of generated members and classes by analyzing the composed superimpositions in their execution order. We propose to record the mapping for every code element in an *index structure*. If the generating transformations are superimpositions and refactorings, a qname (abbr. fully-qualified name) of the generated code should be mapped to a qname of the generating code. Specifically for superimpositions, the qname of a generated method or class in the index should map to an ordered list of qnames in the superimpositions – one for every method/class refinement. Finally, we record the refactorings applied to every superimposition in a *transformation history* (here: refactoring history).

In the index we create from Figure 3, the generated key Queue.first maps to Queue.get from superimposition *Base* and to Queue.get from superimposition *F1*. In the transformation history, *R1* is recorded to affect *Base* and *F1*.

The qname of a code element which got generated by a surjective-typed RFM (from different elements in the generating code) is included in the index but maps to an empty list of qnames.

Plan change propagation. Using *diff*'s result an automated mapper should identify the changed and added code elements (removing elements is not supported generally as we explain later). The mapper then should calculate the transformation to which a change should be propagated (*target*). For that calculation in our sample transformations, we perform two steps: (a) if the element got changed (qname exists in index), we take the superimposition of the indexed value qname as a first target; (b) we analyze the dependencies of the changed/added generated

[12] http://www.gnu.org/software/diffutils/diffutils.html

code toward code elements this code references. Using our index, we calculate the originating superimpositions for these elements (not their refinements) and – to maintain bounded quantification – select the superimposition which was applied last to be the new target.

After we identified the presumably best target superimposition for a change, we check whether this change can be propagated to the target. For that, we check whether all refactorings and superimpositions can be inverted which applied after the target superimposition (recorded in transformation history and index). If all refactorings can be inverted, we propagate the changed element into the target superimposition (possibly replacing the original). If we cannot invert a refactoring of an RFM we provide a *fallback strategy*. As a fallback strategy, we add a new superimposition which succeeds the non-invertible RFM, a superimposition which then encapsulates the debug change. The buggy code will then be overridden in future compositions by the code of the new superimposition.

Refactorings cannot be inverted directly when a debug change violates the inverse refactoring's preconditions [9].[13] To invert a refactoring, we propagate a copy of the changed generated code and apply the inverse refactorings to the copy's code elements important to the inverted refactoring. For example to invert an Encapsulate Field refactoring, we apply changes to the `get` and `set` methods, to the encapsulated field, and the debug change. To invert a Rename Class refactoring, we update the class name along with the debug change elements.

For our sample transformations of refactorings, name capture[14] can prohibit to invert a refactoring. To detect name capture, we extend our index to keep all qnames – even deleted ones (deleted qnames can be tagged with a boolean value). We then can analyze the qname histories for all qnames in superclasses and subclasses and evaluate whether one of them will be newly overridden after change propagation.

If debug changes include multiple new code elements, we define targets for added fields first and organize the changes to constructors relatively to the field targets; methods are the last elements which we define targets for (also relatively to the field targets). Finally, we proof whether bounded quantification got broken and if so we adapt the targets of breaking elements.

When a qname exists in the index but maps to an empty list of value qnames, we infer that the element got generated by an RFM.[15] In this case, we propagate the debug change along the reverse global sequence of program transformations and invert these transformations until one transformation identifies the changed

[13] For example, to generate a program, an Inline Method RFM may have replaced all calls to a method by the called method's body. In the generated code, the according statements now exist as multiple copies – changing one copy prohibits to invert the refactoring.

[14] Name capture redirects method calls and thus changes behavior [34]. Name capture occurs when after a refactoring executed two methods override each other which did not override each other before the refactoring executed.

[15] The superimposition composer also can generate code elements which have no equivalent in the generating code. These elements map to a source file but no value qname in our index.

element as "self-generated" and adapts the target. For instance, a method generated by a Pull-Up Method refactoring from multiple methods occurs within the index but without (generating) value qname. If this method got changed, RFMs are inverted until the refactoring which inverts the Pull-Up Method RFM recognizes the method got generated by the uninversed Pull-Up Method RFM. The inverse refactoring then generates debug changes for every pulled up method. The new debug changes, finally, are propagated to their respective targets.

Save propagation. There commonly are multiple superimpositions a debug change might be propagated to and so advise given to the user might be suboptimal though correct. A mapping tool, thus, should advise a mapping to the user but should ask the user to confirm.

4.2 Unsupported Changes for Superimposition Transformations

The transformations available for program generation limit the changes possible to propagate to the generating code. In some cases even no fallback strategy is available. For example, in superimposition languages (1) the elements, which can be generated, limit the changes that can be propagated and (2) deletions might not be propagated.

Superimposition transformations in Jak do not allow to override constructors [33] so the presented fallback strategy cannot be used when, due to a debug change inside a constructor, the inverting of a refactoring fails. According changes cannot be propagated to the generating code. As a workaround, we propose to follow the advise from the Jak documentation *Extract constructors into initialization methods* [33]. That is, we propose to extract respective constructors automatically into methods which then *can* be overridden. We envision similar workarounds for changed field initializations (fields also cannot be overridden in Jak), by encapsulating their initialization in methods which can be overridden.

Removing elements from the generated code is not supported in every case for superimposition approaches. We can calculate the removed code elements by comparing the qnames of the index with those present in the changed generated code. However, a removed element, then should be removed in the generating code, too. If the debug change prohibits to invert an RFM we cannot delete this code because superimpositions can only generate code.

4.3 Sample Process

In Figure 5, we review the result of our running example of Figure 3 but with debug changes (we underlined changed and added code). We first detect the debug changes. Then we generate the index and the transformation history and map the detected changes to the generating superimpositions. In Figure 5, the first change is detected for a constructor `Queue()` (Line 4) for which a qname *exists* in our index – the code element thus got *changed*. The index value for this qname is a list of qnames – one qname from the superimposition, that introduces

the constructor, and one from the superimposition that refines the constructor.[16] The change is detected in the first block of the constructor and thus the target is the superimposition which introduces Queue(). The second block of Queue() is unchanged and thus there is no need to adapt the constructor refinement. As no dependency toward code elements is added, the target for the detected debug change is the Queue()-introducing superimposition *Base*. We calculate that *R1* (applies after *Base*) can be inverted and advise the user to put the change to *Base*.

A second change is detected in the code of Figure 5 (Lines 9-12) but we do not find a key in our index for the affected qname Queue.flush, i.e., this method got *added* during debugging. To guarantee bounded quantification, we advise to put Queue.flush into a superimposition where all code elements referenced inside flush have been defined before. For that, we analyze the code elements which flush references (Queue.track, Queue.first) and analyze in which superimposition each of them was generated finally: Queue.track is generated in *F1* and Queue.first is generated in *Base* (in *F1* the method just gets refined). We check whether *R1* can be inverted for Queue.flush – it can – and, thus, we add Queue.flush to *F1* which introduced the last referenced code element.

```
1  class Queue{
2   Queue(){
3    {
4    int i =0;
5    }
6    { ... }
7   }
8
9   void flush(){
10   track();
11   first();
12  }
13 }
```

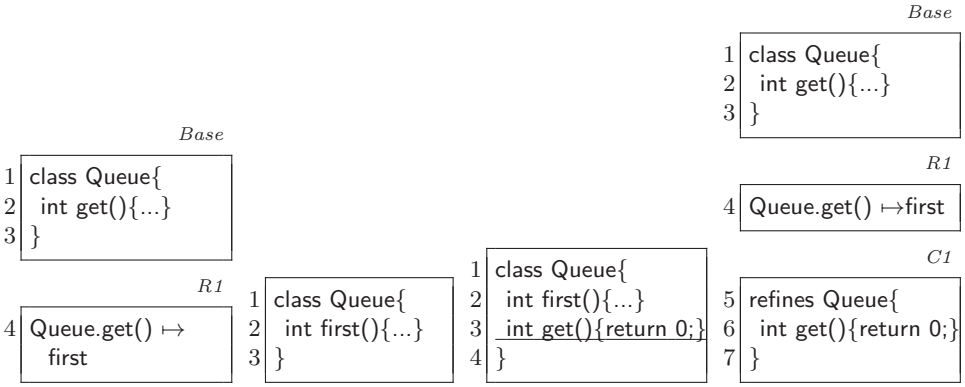
Fig. 5. Found changes in debugged code.

Now let's review an example, where propagating changes does not work. For that, consider a densed version of our running example in Figure 6a. We compare the unchanged code (Fig. 6b) to the code the user changed during debugging. We then create the index and the transformation history. We analyze the first change (Fig. 6c, Line 3) and find it a new method because its qname does not exist untagged in the index.[17] We identify the superimposition *Base* as the best target because the hosting class Queue is defined in *Base* and the added method get has no dependencies to other elements. Following the transformation history of *Base* backwards, we try to invert *R1*, i.e., we try to rename Queue.first into get in the generated code. We fail inverting *R1* due to the added method get because inverting *R1* requires Queue.get to not exist in the code to refactor.[18] As a result, we cannot propagate get from the debugged code into the target *Base*. Following our fallback strategy we propagate Queue.get into a new refinement of Queue in a new superimposition *C1* which follows the non-invertible *R1* (Fig. 6d).

[16] Support for constructors is not yet implemented but the concept is equivalent to the one of supported methods.

[17] The qname does exist in the index because we keep all qnames generated during program generation. However, the key is tagged as deleted, so it cannot match.

[18] In Java, C++, and alike languages, a qname must be unique inside a program [16, p.123ff]. As a result, inverting *R1* must be disallowed to create a second Queue.get.



(a) Feature-oriented Code (b) Unchanged Code (c) Changed Code (d) Decompose Proposal

Fig. 6. A fallback strategy.

5 Prototype and Evaluation

To identify the debug changes our prototype regenerates the debugged product – without debug changes – into a separate folder and compares this code with the debugged code (for datails, see [36]). Our prototype propagates debug changes applied to programs which were generated using the Jak composer tool Jampack [4]. For that, our prototype includes simplifications specific to Jampack – the concept, however, is not affected.[19] The prototype identifies debug changes using a common *diff* tool. To create the index and transformation (refactoring) history, our prototype iterates the superimpositions in their execution order. For every superimposition, it collects the qnames of code elements which are generated in this superimposition as index keys and the according ASTs as index values. If a qname is regenerated the value AST for this qname is replaced, if a method gets refined, we extend the value list of ASTs for the refined qname. If the recorded transformation is an RFM, we execute the refactoring on the keys of our index but not on the index values. Thus, the index in the end maps qnames of the generated code to ASTs of the generating code. Detection of name capture

[19] Jampack generates a method with a mangled name for every method refinement; those methods then call each other according to their former refinement relation. Our prototype's index does not map a generated qname towards a generating qname but towards an *abstract syntax tree (AST)* parsed from the generating code element. Our prototype's index further does not map a method's qname towards a list of according refinement ASTs but maps the qname of each *mangled* method towards its refinement AST. Refinement chains then are computed by following refinement links added for this purpose to qnames. For superimposition approaches that merge refining with refined bodies either the change propagation tool must implement a more detailed comparison to detect the index mapping or the composers must separate refinements differently, e.g., with blocks.

is not yet implemented. After the prototype propagated the debug changes into superimpositions, it saves this advise in a separate folder.

Case study. We evaluate the proposed index solution using the *Graph Product Line (GPL)* which has been proposed a standard study for SPL technology [26]. Specifically, we use a version of prior work in which we applied RFMs to GPL in order to integrate GPL products with incompatible environments [23]. As our prototype implementation is capable yet of calculating the inverse only for Rename Method, Rename Class, and Move Method refactoring, we prune the study accordingly. Furthermore, the prototype can yet propagate only specific changes to the superimpositions: Changes to generated method bodies and additions of methods and fields.

As a proof of concept we selected nine features from the GPL that form a product, five superimpositions and four RFMs. 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. We indicate the executed program transformations in their execution order in Figure 7.

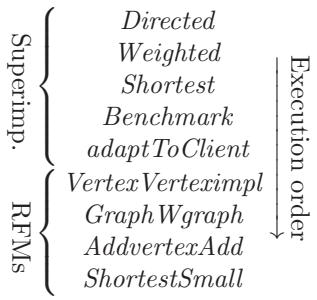


Fig. 7. Studied GPL features.

In Figure 8a, we underline three debug changes applied to the generated class WeightedGraphImpl. The method addEdge(Edge) got *changed* so we compute from our index that addEdge was generated lastly in the superimpostion *Directed* which becomes our first target. We detect that no dependencies towards other code elements were added by the change so *Directed* remains our target. We invert the four RFMs which applied after *Directed* and advise the user to replace the unchanged method in *Directed*. The second change detected, concerns a method ShortestPath which was *added* during debugging. We analyze the references in this method – ShortestPath solely references shortestPath (generated in superimposition *Shortest*) so superimposition *Shortest* is our first target. As a next step we check whether all RFMs that applied after *Shortest* can be inverted with ShortestPath. We cannot invert RFM *ShortestSmall* (renames WeightedGraphImpl.ShortestPath(VertexImpl) into shortestPath) because this would create two methods with the qname WeightedGraphImpl.ShortestPath(VertexImpl) in the generating code. We add a superimposition as a successor of *ShortestSmall*, a superimposition which then encapsulates the debug change ShortestPath.

In Figure 8b, we highlight the debug changes applied to class VertexImpl. The field displayed and the method wasDisplayed got added; and methods display and assignName got changed to use the added field and method. We detect those uses and for that advise to use superimposition *Shortest* (generates display) as a target for displayed. As wasDisplayed got added and solely references displayed, it

(a)

```
1  class WeightedGraphImpl{ ...
2   public void addEdge(Edge the_edge) {
3    if (the_edge != null) {
4      VertexImpl start = the_edge.start;
5      VertexImpl end = the_edge.end;
6      edges.add(the_edge);
7      start.addNeighbor(new Neighbor(end, the_edge));
8    }else{
9      System.out.println("Param the_edge was null!");
10   }
11  }
12  public WeightedGraphImpl ShortestPath(VertexImpl s) {
13   return shortestPath(s);
14  }
15 }
```

(b)

```
16 class VertexImpl{ ...
17  private boolean displayed = false;
18  public void display() {
19   System.out.print("Pred " + predecessor + " DWeight " + dweight + " ");
20   display$$eval$outWeighted$GG();
21   this.displayed = true;
22  }
23  public boolean wasDisplayed(){
24   return displayed;
25  }
26  public VertexImpl assignName(String name) {
27   this.name = name;
28   if(this.wasDisplayed()){
29    System.out.println("was already displayed!");
30   }
31   return (VertexImpl)this;
32  }
33 }
```

(c)

```
34 class Edge { ...
35  public void EdgeConstructor(VertexImpl the_start, VertexImpl the_end) {
36   if(!the_end.equals(the_end)){
37     EdgeConstructor$$eval$outDirected$GG(the_start,the_end);
38   }
39  }
40 }
```

Fig. 8. Detected changes in the GPL case study.

is advised to be put to *Shortest*, too. To keep bounded quantification, the changes to methods assignName (generated in the early superimposition *Directed*) and display are advised to be put into *Shortest*, too.[20]

Yet, our prototype does not support refactoring transformations which establish partial mappings (0↦1). So we evaluated the proposed concept using a method which is generated by Jampack during superimposition execution, i.e., a method which does not have a generating equivalent in the superimpositions. In Figure 8c, we show a method EdgeConstructor(VertexImpl,-VertexImpl) of class Edge which was generated by Jampack because method Edge.EdgeConstructor(Vertex,Vertex,int) in the superimposition *Weighted* called this method with an explicit Super call.[21] That is, to this method no generating code exists in the superimpositions. When we created the index, we detect that Jampack will generate this method and added its qname to the index (without value AST). As a result our prototype advises to put the transformed method refinement as a method refinement to the superimposition *Weighted*. As a result, Jampack will not regenerate the method and the debug changed code equals the regenerated code.

6 Related Work

Mapping to higher-level languages. Preprocessor directives are common to the C++ language [35, p.606ff]. Line directives, added to the generated code, indicate from which file and which source line the following lines in the generated code stem from.[22] In Java, users can create file- and line-based mapping tables (called *source maps (SMAPs)*) and optionally compile them into binaries [37].[23] We refrained from using these techniques because preprocessor directives and SMAPs are not available for every language and not every debug tool may understand them.

Comments are available in most programming languages. Comments can be added to the generated code to encode the origin of generated code lines that follow. Comments are language independent but commonly are not integrated into generated binaries. A debug tool would thus have to synchronize the binaries, the generated code, *and* the generating code. In Java, comments can be

[20] In this case, it would have been better to propagate displayed and wasDisplayed to earlier superimposition *Directed* because the changes to method assignName could then be propagated to *Directed* (generates assignName) and the changes to display could then be propagated to *Shortest* (generates display) without violating bounded quantification. Such considerations are possible future work.

[21] Jampack generates a method A in the code, when a method B calls A of B's *respective* preceding superimposition using Super, i.e., B does not refine A but calls an intermediate A.

[22] Line directives are used for debugging in the FOP language FeatureC++ [3]. Line directives are used to debug C++ template metaprograms [31].

[23] SMAPs are used for debugging in AspectJ (see http://eclipse.org/aspectj/sample-code.html#trails-debugging-aspectj10, http://eclipse.org/aspectj/doc/released/faq.php).

compiled in annotations [16, p.281]. Finally, to maintain comments, numbers of existing SPL composer tools would require adaptation.[24]

The DSL Debugging Framework (DDF) integrates mapping information into DSL grammars [39]. DDF maps one line in the generating code to a sequence of lines in the generated code (surjective mappings are not considered). Thereby, DDF maps a DSL line number onto the accordingly (a) generated file, (b) range of line numbers, (c) the function of the mapped DSL line, and (d) the type of the mapped DSL statement.

KHEPERA tracks the effects of transformations which are successively applied to a program and generates code which implements the mapping [13]. To use KHEPERA, the generator must integrate with KHEPERA to trigger AST-change-logs; or the transformations must be written in the KHEPERA language (then it is translated by KHEPERA and logs are triggered, too). Integrating composers with KHEPERA might not be possible when composers are legacy applications. KHEPERA allows to view and step in intermediate transformation results [12] but does not propagate changes from the generated code towards the generating code.

Beside SMAPS, all above approaches would require extensions to existing Jak and RFM composer tools. In contrast, our prototype works alongside the Jampack composer tool and the RFM composer tool and does not impose changes to them. Furthermore, the index-based approach supports surjective mappings which are unsupported by all above approaches except for KHEPERA.

Debugging advances for metaprograms. Aspect transformations of aspect-oriented programming are similar to superimposition transformations [25]. Eaddy et al. report on unexpected and confusing code jumps when aspect transformations are hidden during debugging [10]. For such situations they propose to display the injected and interwoven aspect code. When a bug in the injected and interwoven code is found, the user must change the generating aspect – they do not focus on *how* to identify the generating (aspect) code to change. Ishio et al. calculate a call graph that relate aspect code and extended code in order to calculate erroneous loops and accidental advice execution [20]. They further slice programs to expose the source statements that lead to an incorrect variable value. Ishio et al. do not argue how to change the program or the aspects. Aspect transformations differ from refactorings.

Porkoláb et al. show how they instrument C++ meta metaprograms, which should be debugged, and record their execution [31]. Based on the records they subsequently step in the template code but they cannot influence the (priorly) executed program. In contrast to them, we aim at debugging the generated runtime program rather than the generating compile time program. Further,

[24] In the Unmixin tool [4] an annotation-based approach (SoUrCe declarations) is used to map generated Java code to Jak code of superimpositions. In addition to propagating debug changes to superimpositions (as performed by Unmixin), our index concept supports refactorings as SPL transformations. Refactoring transformations are not supported by Unmixin.

tracing a large-scale program statement-by-statement (to allow statement-based stepping) might not be feasible.

Debugging optimized code. Compilers apply refactoring-like transformations to code such that the transformed program executes faster than the original. Debugging these programs pose similar problems as we described. The problem of merging variable values in Section 3.1 highly correlates to the problem of non-current and endangered variables after program optimizations [17,1]. The difference is that, in our case, different stores may replace a single store without synchronization intention, i.e., they may expose different values without changing behavior. Path analyses, a proposed solution in prior work, is hard (if not impossible) to implement for source-to-source transformation systems, like Jak and RFM tools. Further, the current value of the generating field might be independent from the latest assignment, e.g., it may depend on the subclass code, such that "simple" path analyses could not help.

The problem of displaying an ambiguous copy, and of breakpoints matching too often or too seldom can be solved by path analyses and conditional breakpoints [40][1, p.147]. This however, might be hard (if not impossible) for source-to-source transformation systems or certain target languages.

In contrast to the work on debugging optimized code, the transformations considered here are not only restructuring operations but also functionality-adding transfomrations of superimpositions. This code exists in the generated program but not in the base program to which superimpositions and refactorings apply. The approaches discussed above would impose changes to existing composition tools.

Debug perspective. Some researchers argue to display the generating code to the user, e.g., [39,18], and some researchers argue that debugging should be possible at every level of abstraction [6, p.326] [13,20,11]. When SPLs include complex code transformations like refactorings and the language of the generated code is similar to the language of the generating code, we argue to debug the *generated* code and assist the user in propagating the debug changes afterwards to the SPL transformations. The reasons are manifoldly: First, the complex mapping of sequenced refactorings (cf. Sec. 3.1) hampers to step comprehensibly through the generating code. Second, current development tools of superimposition-based SPLs require the user to compose programs in mind and do neither hide code unavailable nor highlight code available in the debugged product (cf. Sec. 3.2). Third, tools to debug the generating code can hardly assist the user in which superimposition to put a debug change best to minimize complexity (cf. Sec. 3.3). Finally, the language of the generating code (e.g., Jak) and the language of the generated code (e.g., Java) are highly similar in superimposition and refactoring-based SPLs such that there should be no adaptation problems of displaying the generated code.[25]

[25] Jak is Java plus 3 keywords, Super, refines, and Layer [5].

Inverting refactorings. MolhadoRef inverts refactorings to reduce conflicts when merging a debug-changed program with a former revision of this program [9,8]. Inline with MolhadoRef we execute inverse refactoring operations in order to invert refactorings. In contrast to MolhadoRef, the refactoring sequence we invert was not executed during debugging but during program generation. While for MolhadoRef the differences between the program and the code base are identified and handled *after* inverting refactorings, we have to identify debug changes *before* inverting refactorings in order to propagate the debug changes correctly (to identify target superimpositions). While MolhadoRef propagates debug changes of a program toward a single program (an earlier revision), we propagate debug changes of a program toward a code base of a transformation-based SPL.

In the context of compiler optimizations, transformations, like Inline Method, are inverted at runtime to debug the generating code [18]. Our setting is the debugging of a product of a transformation-based SPL and not the debugging of an optimized off-the-shelf program. Refactorings are just one kind of transformation we invert. [18] does not describe how to deal with surjective transformations.

Bidirectional transformations. Bidirectional transformations (a.k.a. lenses) synchronize multiple related representations of elements where changes can be triggered in any representation [7,19,14,30]. The common generation process is described as transforming a higher-level representation (abstract view) into a lower-level representation (concrete view). We can think of superimposition code being a concrete view and the generated code being a concrete view, too – the change propagation we perform then is inverting RFMs and detaching method refinements. Our index structure, thus, can be interpreted as being the abstract view. For that, however, ASTs of the generating and generated code must be integrated because then we could generate the concrete view (program) from the abstract view (qname index). In the change propagation problem we focused on, edits to the generated code may prohibit the application of inverse RFMs (we, thus, discussed a fallback strategy) – such situation may not occur for bidirectional transformations.

7 Conclusions

In this paper we discussed a number of problems which occur when debugging a program generated from a transformation-based software product line (SPL). Specifically, we discussed problems of complex mappings between the generating and the generated code, problems of generating SPL products in mind, and problems of debug changes breaking the SPL design. We found that for SPLs implemented with transformations of superimpositions and refactorings, the *generated* code should be debugged and corrected. After that, the developer should be *assisted* in propagating the debug changes to the generating transformations. We presented the concepts needed to assist the developer in propagating changes and evaluated them using a prototype.

Acknoledgments

The authors thank Don Batory, Christian Kästner, and Marko Rosenmüller for helpful comments on earlier drafts of this paper.

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, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.
3. S. Apel, M. Rosenmüller, T. Leich, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140, 2005.
4. D. Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 3–35, 2006.
5. D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.
6. K. Czarnecki and U. Eisenecker. *Generative programming: Methods, tools, and applications*. Addison-Wesley, 2000.
7. K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283, 2009.
8. D. Dig. *Automated upgrading of component-based applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.
9. D. Dig, K. Manzoor, R.E. Johnson, and T.N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering (TSE)*, 34(3):321–335, 2008.
10. M. Eaddy, A.V. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Proceedings of the International Symposium on Software Composition (SC)*, pages 200–215, 2007.
11. C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat. Variability in time – product line variability and evolution revisited. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 131–137, 2010.
12. R.E. Faith. *Debugging programs after structure-changing transformation*. PhD thesis, University of North Carolina at Chapel Hill, 1998.
13. R.E. Faith, L.S. Nyland, and J.F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, pages 19–19, 1997.
14. J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 2005.

15. M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.
16. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification.* Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2005.
17. J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):323–344, 1982.
18. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27(7):32–43, 1992.
19. Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 178–189, 2004.
20. T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 178–187, 2004.
21. 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, Software Engineering Institute, Carnegie Mellon University, 1990.
22. C. W. Krueger. New methods in software product line practice. *Communications of the ACM (CACM)*, 49(12):37–40, 2006.
23. M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 106–115, 2009.
24. M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 177–186, 2009.
25. M. Kuhlemann, M. Rosenmüller, S. Apel, and T. Leich. On the duality of aspect-oriented and feature-oriented design patterns. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 5, 2007.
26. R.E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24, 2001.
27. R.E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2005.
28. M. Odersky. *The Scala language specification (version 2.7)*, 2005.
29. W.F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
30. B.C. Pierce. Foundations for bidirectional programming. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 1–3, 2009.
31. Z. Porkoláb, J. Mihalicza, and Á. Sipos. Debugging C++ template metaprograms. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 255–264, 2006.
32. D.B. Roberts. *Practical analysis for refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.
33. Software Systems Generator Research Group. *The jampack composition tool.* AHEAD tool suite v2008.07.22, manual.
34. P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285, 1996.

35. B. Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 1991.
36. M. Sturm. Debugging Generierter Software nach Anwendung von Refactorings. Master thesis, University of Magdeburg, Germany, MAR 2010.
37. Sun Microsystems, Inc. *JSR-000045 Debugging support for other languages 1.0 FR*, 2003.
38. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104, 2007.
39. H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10):1073–1103, 2008.
40. P.T. Zellweger. An interactive high-level debugger for control-flow optimized programs (summary). In *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on high-level debugging*, pages 159–171, 1983.