Genetic Improvement for Software Product Lines: An Overview and a Roadmap

Roberto E. Lopez-Herrejon[1], Lukas Linsbauer[1], Wesley K. G. Assunção[2,3],
Stefan Fischer[1], Silvia R. Vergilio[2], Alexander Egyed[1]

[1]ISSE - Johannes Kepler University Linz, 4040 Linz, Austria
[2]DINF - Federal University of Paraná, CP: 19081, 81531-980, Curitiba, Brazil
[3]COINF - Technological Federal University of Paraná, 85902-490, Toledo, Brazil
{roberto.lopez, lukas.linsbauer,stefan.fischer, alexander.egyed}@jku.at,
{wesleyk, silvia}@inf.ufpr.br

ABSTRACT

Software Product Lines (SPLs) are families of related software systems that provide different combinations of features. Extensive research and application attest to the significant economical and technological benefits of employing SPL practices. However, there are still several challenges that remain open. Salient among them is reverse engineering SPLs from existing variants of software systems and their subsequent evolution. In this paper, we aim at sketching connections between research on these open SPL challenges and ongoing work on Genetic Improvement. Our hope is that by drawing such connections we can spark the interest of both research communities on the exciting synergies at the intersection of these subject areas.

CCS Concepts

•**Software and its engineering** → **Software product lines; Search-based software engineering;** •**Computing methodologies** → *Search methodologies;*

Keywords

evolutionary algorithms, genetic programming, genetic improvement, software product lines, variability

1. INTRODUCTION

Software Product Lines (SPLs) are families of related software systems [5], where each product has a different combination of *features* – increments in program functionality [48]. Over the last two decades, extensive research and application of SPL practices have shown their technological and economical advantages (e.g. [46]). However, for an effective adoption of SPLs many challenges still need to be addressed. Salient among them are the reverse engineering of SPLs from existing software system variants, the most prevalent scenario in industry [10], and the evolution of SPLs [3, 7, 23].

Genetic Improvement (GI) is the process of automatically improving a system's behavior using genetic programming [24]. In contrast with typical work on genetic programming that evolves programs from scratch [41], GI starts from existing systems which are evolved so that they are better with respect to given criteria. GI is an expanding research area that has already been successfully used, for instance, to specialize boolean satisfiability (SAT) programs written in C++ for concrete tasks such as combinatorial interaction testing [40], to repair broken functionality [15], or to add new functionality [18]. In recent publications, Harman and colleagues advocate the potential use of GI for SPL development [17, 24]. For instance, to generate a Pareto surface of programs that exhibit the same functionality but with different quality attributes such as performance. Another possibility they propose is actually growing new functionality, i.e. creating a new branch or product variant of a SPL.

Our recent systematic mapping study showed the increasing interest in research at the the intersection of SPLs and Search Based Software Engineering (SBSE) [34]. The driving motivation of this paper is to sketch in further detail the connections between SPLs (reverse engineering and their evolution) and ongoing work on GI. We believe that by drawing these connections we could bring the attention to the similar challenges faced by both research communities, and draft a rough roadmap of open issues that could be addressed in the short to medium term. We hope this paper can help to spark the interest on the exciting synergies at the intersection of these areas.

The paper is organized as follows. Section 2 provides the basic background to understand the SPL challenges that our paper focuses on. Section 3 draws the connections between ongoing work on SPLs and the work of GI as characterized by the *Genetic Improvement of Software for Multiple Objective Exploration (GISMOE)* project [19, 24]. Section 4 presents our roadmap of issues that could be addressed as some of the first steps in exploring the synergies between SPLs and GI.

2. SPL BACKGROUND

Some of our recent work has focused on several novel approaches for reverse engineering SPLs and their subsequent evolution [1, 2, 12, 13, 27, 28, 33, 35]. In this section, we illustrate the two main challenges that our work addresses.
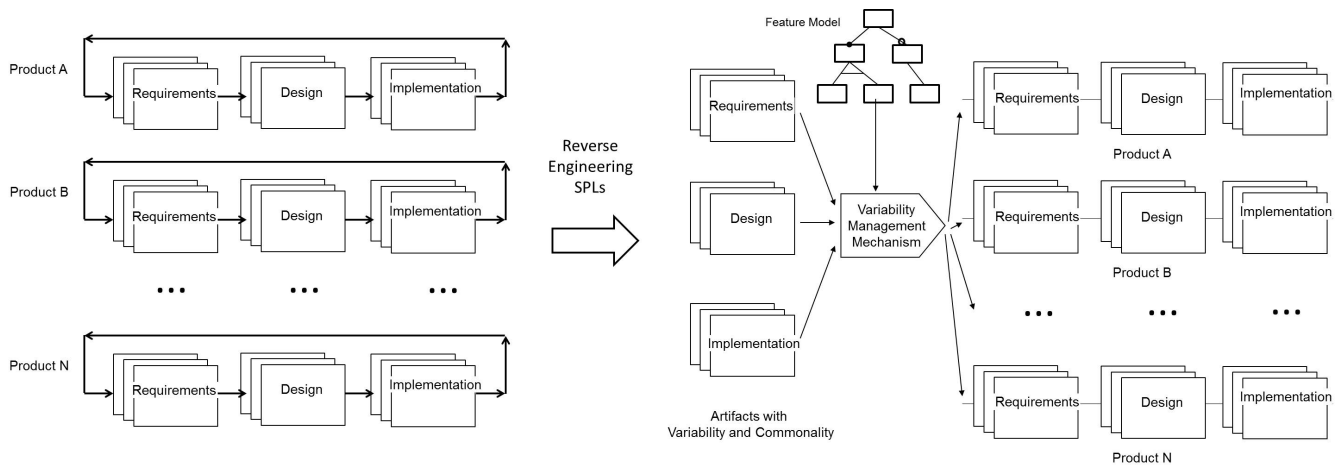
Figure 1: Reverse Engineering Software Product Lines Overview

2.1 Reverse Engineering SPLs

The most common scenario for reverse engineering SPLs in practice is depicted on the left part of Figure 1. In this scenario, there exist several related software products (i.e. they offer some similar functionality) that were created partially independently (e.g. they may have shared some artifacts at some point in their history) typically with ad hoc techniques; for instance, by forking the new products from other already existing software systems and then manually adapting them to fit the new requirements. Such products follow their own, mostly independent, development life cycle. Ad hoc practices like these are collectively called *Clone and Own (C&O)* [10], and support available for them is very limited.

Not surprisingly, even when dealing with a small number of products, C&O approaches lead to multiple maintenance problems like inefficient feature update or bug fixing (i.e. a bug needs to be fixed separately for each product), duplicate functionality, redundant and inadequate testing, etc [10]. In other words, C&O approaches may work fine up to a certain number of distinct products depending on product domains, the development organization and its software engineering practices. However, after such number, simply adding new independently-developed systems is no longer feasible either because of managerial, economical or technical reasons.

It is then that SPL approaches are the premier alternative for effectively coping with the complexity of the *variability* – the capacity of software artifacts to vary— in the existing products. The transition to a SPL approach is not a minor undertaking because it requires a significant investment of resources to identify, extract, and reify what is common (i.e. commonality) and what varies (i.e. variability) across all the artifacts involved as depicted on the right part of Figure 1. The artifacts with their variability and commonality identified constitute the building blocks used by variability management mechanisms to generate the existing products but should in addition be capable of providing robust support for maintenance and evolution tasks. There are many variability management mechanisms, all of them with advantages and disadvantages (see e.g. [14]). What is common among variability management mechanisms is that they rely on a *variability model* to express the valid combinations of features desired for the products of a SPL. The most popular form of variability models is *feature models* depicted at the top of Figure 1. For further details please refer to [21]. Our previous work has explored the use of SBSE techniques for reverse engineering feature models [2, 28, 33, 35].

In this paper, the focus is on reverse engineering the variability and commonality of the artifacts that are fed into the variability management mechanisms. We use as a running example simple software products that draw different geometric shapes on a screen canvas. Even though our example uses Java code for illustration, the challenges that will be highlighted can be easily mapped to other types of artifacts.

In SPLs, each software product is distinguished by the set of features it provides. Let us consider an example of a product `P1` which provides a single feature `LINE`. This product consists of two classes whose code snippets are shown in Figure 3. Class `Line` contains fields for the start and end points of lines (Line 3), a constructor (Line 4), a method `paint` that paints the lines (Lines 5-8) and an auxiliary method `setEnd` that sets the end point (Line 9). Class `Canvas` contains field `lines` that holds a list of the lines drawn (Line 12), and method `paintComponent` (Lines 13-18) which is called every time the canvas is repainted and that in turn draws the lines on the canvas by calling their method `paint` (Line 16).

Now consider a second product `P2` whose code is also sketched on Figure 3. In addition to feature `LINE`, this product also implements feature `WIPE` that wipes the canvas clean. This new feature causes changes in the code because it adds a new method `wipe` (Lines 38-41) shown underlined in the figure. When this method is called, the list that holds the `Line` objects is cleared (Line 39), so that when the call to `repaint` (Line 40) is made, the triggered call of `paintComponent` (Line 32) finds no lines to draw in Line 35.

Let us consider a third product `P3` that implements feature `RECT` which draws rectangles on the canvas. The code snippets are also shown on Figure 3. Similarly to variant `P1`, it defines a new class, `Rect` (Lines 44-52), to implement the new shape type. This class contains the fields to draw the rectangle, upper left corner coordinates plus width and height (Line 45), a `paint` method (Lines 47-50), and a `setEnd` method (Line 51). This variant also contains its `Canvas` class (Lines 53-61), which now has a list of rectangles

(Line 54) and its method `paintComponent` (Lines 55-60) that paints the rectangles (Line 58).

Based on these three products of our running example, the goal is to reverse engineer a SPL that we will refer to as the *Draw Product Line (DPL)*. As mentioned before, there are many variability management mechanisms that have been used to implement SPLs. The simplest ones are based on annotations made to the artifacts that are pre-processed based on the different selection of features for each product that needs to be created. Figure 2 shows an example of how the class `Canvas` could be annotated. In this simple example, a product that contains feature `LINE` will include in the final code of class `Canvas` the pieces of code of Line 3 and Line 12. Similarly for products that provide feature `RECT` which will include Line 6 and Line 15. In practice, the conditions that guard the inclusion of the fragments of these artifacts can be more elaborate to reflect the complex interactions that features can have. This will be elaborated on in Section 3.1.

The main challenge for reverse engineering SPLs is thus to automate as much as possible the transition from the artifacts of individual products to the corresponding artifacts that, together with a chosen variability management mechanism, correctly and completely capture all possible and valid combinations of features. This challenge becomes more daunting when hundreds of products, hundreds (if not thousands) of features, and multiple artifacts other than source code are considered.

```
1    class Canvas {
2     #ifdef $LINE
3     List<Line> lines  = new LinkedList<Line>();
4     #end
5     #ifdef $RECT
6     List<Rect> rects  = new LinkedList<Rect>();
7     #end
8
9     void paintComponent(Graphics g)   {
10        ...
11      #ifdef $LINE
12      for (Line l : lines) { l.paint(g); }
13      #end
14      #ifdef $RECT
15      for (Rect r : rects) {  r.paint(g); }
16      #end
17       ...
18     }
19   }
```

Figure 2: Example DPL using preprocessor annotations

2.2 Evolution of SPLs

Broadly speaking we define SPL evolution as the progression of the changes made to a SPL towards a better adaptation to the environment where it is intended to be used. There are multiple evolution scenarios for SPLs [26]. For illustration purposes here we focus on a single one, the creation of a new product based on the combinations of existing features. More concretely, let us now consider the case of developing a new product P4 that provides both rectangles (`RECT`) and wiping capabilities (`WIPE`), both features already available in the SPL from two previous products P2 and P3. The challenge now becomes how to build the new product based on the knowledge already available, as captured, for instance, in the pre-processor implementation of Figure 2.

```
1  /* Product 1 (LINE) */
2  class Line {
3     Point startPoint, endPoint;
4     Line(Point start) {...}
5     void paint(Graphics g) {
6        g.setColor(Color.BLACK);
7        g.drawLine(startPoint.x, startPoint.y,
              endPoint.x, endPoint.y);
8     }
9     void setEnd(Point end) {...}
10 }
11 class Canvas {
12    List<Line> lines = new LinkedList<Line>();
13    void paintComponent(Graphics g)  {
14       ...
15       // Paints the figures
16       for (Line l : lines) { l.paint(g); }
17       ...
18    }
19 }
```

```
20 /* Product 2 (LINE, WIPE) */
21 class Line {
22    Point startPoint, endPoint;
23    Line(Point start) {...}
24    void paint(Graphics g) {
25       g.setColor(Color.BLACK);
26       g.drawLine(startPoint.x, startPoint.y,
              endPoint.x, endPoint.y);
27    }
28    void setEnd(Point end) {...}
29 }
30 class Canvas {
31    List<Line> lines = new LinkedList<Line>();
32    void paintComponent(Graphics g)  {
33       ...
34       // Paints the figures
35       for (Line l : lines) { l.paint(g); }
36       ...
37    }
38    void wipe() {
39       lines.clear();
40       repaint();
41    }
42 }
```

```
43 /* Product 3 (RECT) */
44 class Rect {
45    int x, y, width, height;
46    Rect(int x, int y) {...}
47    void paint(Graphics g) {
48       g.setColor(Color.BLACK);
49       g.drawRect(x, y, width, height);
50    }
51    void setEnd(int newX, int newY) {...}
52 }
53 class Canvas {
54    List<Rect> rects = new LinkedList<Rect>();
55    void paintComponent(Graphics g)  {
56       ...
57       // Paints the figures
58       for (Rect r : rects) { r.paint(g); }
59       ...
60    }
61 }
```

Figure 3: Example products P1, P2, P3

```
1  /* Product 4 (RECT, WIPE) */
2  class Rectangle {
3      int x, y, width, height;
4      Rectangle(int x, int y) {...}
5      void paint(Graphics g) {
6        g.setColor(Color.BLACK);
7        g.drawRect(x, y, width, height);
8      }
9      void setEnd(int newX, int newY) {...}
10 }
11 class Canvas {
12     List<Rectangle> rectangles = new LinkedList<
           Rectangle>();
13     void paintComponent(Graphics g)  {
14         ...
15         // Paints the figures
16         for (Rectangle r : rectangles) {
17             r.paint(g);
18         }
19         ...
20     }
21     void wipe() {
22         lines.clear(); //faulty feature interaction
23         repaint();
24     }
25 }
```

Figure 4: Example of naive implementation of product P4 (RECT,WIPE).

As a starting point, artifacts from variant P3 can be reused because it already provides the feature that draws rectangles RECT. In addition, based on the implementation of products P1 and P2, it could be inferred that feature WIPE is implemented by the code in Lines 38 to 41 of variant P2, shown underlined in Figure 3, because it is the only part that changes between products P1 and P2.

A naive approach would be to compose features RECT and WIPE as shown in Figure 4. Notice that in this figure, in Line 22 the statement *lines.clear*(); makes a reference to field lines which is not defined in class Canvas of product P4, hence causing an error. This error highlights the presence of a *feature interaction* – code that exists whenever two or more features appear together in a product [22] – between WIPE and LINE, and the presence of code that does not change (e.g. Lines 21, 23, 24) regardless of what other features are present in a product – code which is needed for a valid feature implementation (e.g. WIPE).

The challenge for this evolution scenario is to automatically detect errors like this one, repair them, and update the SPL accordingly, for instance, by modifying the guard conditions of the pre-processor code for the corresponding artifacts. In this example, a simple repair would be substituting Line 22 with *rectangles.clear*().

3. SPLS MEET GISMOE

In this section we start drawing the connections, albeit with broad strokes, between recent and ongoing research on reverse engineering SPLs and their evolution with the Genetic Improvement of Software for Multiple Objective Exploration (GISMOE) approach [19, 20, 24]. What drove our interest into the subject was the following quote presented at a keynote talk at ASE 2012 [19]:

"The GISMOE approach may also offer solutions to some of the issues raised by SPLs. For example, using GISMOE, we can create new branches automatically: the GP engine will evolve the new versions of the product family from existing members of the family. We may also be able to merge versions when the product family becomes large or unwieldy."

Inspired by this quote, in the next subsections we make an attempt at aligning the two challenges described in Section 2 with recent GISMOE publications.

3.1 Feature-level functional sensitivity analysis

GISMOE advocates the use of sensitivity analysis to determine the places in code artifacts that should be evolved and which should remain untouched according to the property analyzed. We argue that to address the SPL challenges with GISMOE, a key premise is to be able to perform a sensitivity analysis at the feature level because features are the building blocks of the products of a SPL. This entails establishing traceability links between features and feature interactions and the artifacts that realize them. These links would enable targeting the evolution to concrete parts of the artifacts to achieve the desired functionality goals.

We now describe this form of sensitivity analysis based on our traceability work [27], which is part of our *Extraction and Composition for Clone-and-Own (ECCO)* approach [12, 13, 25]. ECCO works under two simple premises. First, that each product provides its artifact base (e.g. Java source code) and a list of features that it implements (e.g. feature LINE for product P1, and features LINE and WIPE for product P2). Second, that features with the same name implement the same functionality in similar ways. For instance, in our running example, feature LINE has the same functionality in both products P1 and P2. In addition, for sake of illustration, LINE is shown in Figure 3 implemented with exactly the same code in products P1 and P2; however, we will discuss later the implications of this assumption.

Let us now define the two kinds of *modules* in the software products that ECCO considers [12, 27]:

DEFINITION 1. **Base Module.** *A* base *module implements a feature regardless of the presence or absence of any other features and is denoted with the feature's name written in lowercase.*

DEFINITION 2. **Derivative Module.** *A derivative module* $m = \delta^n(c_0, c_1, ..., c_n)$ *implements feature interactions, where* c_i *is* F *(if feature* F *is selected) or* ¬F *(if not selected), and n is the order of the derivative.*

ECCO's extraction algorithm automatically produces a set of traces between both types of modules and the artifacts (e.g. source code fragments) that implement them [27]. These code fragments can be of any granularity level, from entire classes down to individual statements. In essence, this algorithm works by incrementally refining the traces based on integrating the knowledge that can be derived from each product. Let us illustrate the gist of this idea with the code of Figure 3. As a first step, if we initially consider only product P1, all its code would trace to module line. With the knowledge added by product P2, the algorithm can infer now, based on the different features provided by the two products, that all the code in Lines 21-29, Lines 30-37, and Line 42 traces to module line. This is so because both products, on one hand, provide the same feature LINE

and, on the other, implement this feature in a similar way. In addition, the algorithm now traces Lines 38-41 to module `wipe` and derivative module $\delta^1(\text{line},\text{wipe})$. Note that if we were to add another product with feature `WIPE` but without feature `LINE`, the algorithm could further refine the traces to indicate that the feature interaction denoted by derivative $\delta^1(\text{line},\text{wipe})$ is actually realized by the statement in Line 39. This short description omitted many relevant details such as dealing with hierarchy and ordering issues among the artifacts. For further information, please refer to [12, 13, 27].

The relevance for GISMOE of the traceability provided by ECCO is that this new form of sensitivity analysis provides the means by which to focus the places where to target the evolution of the artifacts, for instance to graft new functionality (i.e. grow and insert a new feature), repair a bug (i.e. the faulty feature interaction in Figure 4), or improve a non-functional property of a feature.

Nonetheless, there remain several challenges that still need to be addressed to fully support this form of sensitivity analysis. Currently, the similarity among artifacts is computed purely on the structural comparison of their elements. We envision that clone detection technologies will play an important role addressing this need (e.g. [6, 42, 43]).

3.2 Automated test case generation

In GISMOE, the test cases are used to evaluate the fitness of the artifacts that result from evolution. There is an extensive body of work on SPL testing [8, 9, 11]. Among the most popular approaches for SPL testing is *Combinatorial Interaction Testing (CIT)*, a fact that has been corroborated by our recent mapping study [32]. However, to the best of our knowledge, the automated test generation for the scenarios described in Section 2 remains an open question.

ECCO was conceived from the outset to be able to support any type of artifact provided that it can be differentiated, i.e. that a mechanism is available for identifying the common and different parts of the artifacts based on given criteria (e.g. artifact structure). From this perspective, test artifacts (e.g. models or code) are simply another type of artifact for which their traceability to features and feature interactions, as described in Section 3.1, can be obtained. Consequently, we envisage that the new test cases can themselves be derived and genetically improved in the same way as other artifacts can, i.e. by focusing on the precise artifact fragments identified by ECCO's traceability algorithm. In addition, we speculate that work on computing CIT covering arrays applied to SPLs (e.g. [31,32]) that instead focuses on the coverage of base and derivative modules could serve to reduce the testing effort, in a similar way to the notion of output bins that has been employed by GISMOE [24].

3.3 Feature-level non-functional sensitivity analysis

Research on non-functional properties in SPLs has attracted recent interest. For instance, Noorian et al. propose a taxonomy to classify the different approaches [38]. A first line of research is the work by Siegmund et al. who estimate memory footprint and main-memory consumption [45]. They propose an analytical model that computes, from a CIT perspective, covering arrays of different strengths (i.e. t=1, t=2, and ad hoc) which they use to perform actual measurements of a single non-functional property. Based on the

measurements, their model estimates the impact that features and feature interactions can have on the non-functional property.

A second line of research works by adding additional attributes to the features of the feature model that models the variability of a SPL. Based on these values, multi-objective evolutionary algorithms are used to obtain configurations (i.e. products with valid combinations of features) that meet the property requirements. For instance, the work of Sayyad et al. focuses on information related to cost and defects [44]. Our previous work considered also multi-objective evolutionary algorithms but instead applied to relevant properties (i.e. usability, battery consumption, and memory footprint) for the dynamic configuration of SPLs deployed in mobile phone environments [39]. In both cases, the evaluation was performed basically with synthetic values.

We argue that the solution to achieving the goal of non-functional sensitivity analysis advocated by GISMOE when applied in the context of SPLs lies somewhere at the intersection of these two lines of research. On one hand it is important that the optimization is based on actual measured values at the right level of granularity (i.e. not at the coarse grain level of attributes of single features) and that the optimization considers multiple objectives, hence multiple non-functional properties, simultaneously.

3.4 The need of co-evolution

The GISMOE approach relies on co-evolution of the programs and their corresponding tests. SPLs make it a more challenging task because, in a real-life scenario, multiple types of artifacts must be concurrently considered and hence co-evolved. In addition, a key requirement is to keep the artifacts not only for a single product but simultaneously for *all* the products of the SPL. Work on consistency checking across multiple variants (e.g. [29]) could be brought to bear as a starting point to address this need.

3.5 The human in the loop

GISMOE advocates the involvement of developers of the systems, in part to employ domain knowledge perhaps only available in the heads of the domain experts but also to help reduce the adoption barrier of its automatically generated systems.

Our ECCO approach also benefits from the involvement of developers. ECCO provides hints, for instance, to indicate that the implementation of some new modules cannot be fully automated [12, 13, 25]. Based on the input received manually, ECCO updates and refines its traces.

Another source of knowledge could be, for example, employing ontologies to provide information about the relationships between features in a given domain. For instance, in our running example, knowing that a `LINE` and a `RECT` are both geometrical forms could help to trim the search space for finding a repair to the faulty product `P4`. More concretely, to use the code of product `P3` as source to obtain code fragments (i.e. field `rects`) from which to evolve or graft the repair.

One of the most salient challenges here is how to convey the large amount of information to the human, such that the interactions are both simple and efficient. Research and tooling available for software visualization is definitively the starting point [47]. Except for few examples, such as [30,36], SPL visualization remains an area largely unexplored.

4. THE ROAD AHEAD

In this section we broadly describe some of the first exploration steps we plan to undertake in the short-term future.

- *Exploit clone detection.* We plan to extend our traceability algorithm to detect similar and different code beyond the current level that focuses on identical names and structure [6, 42, 43].

- *Grafting variability annotations into UML models.* We plan to explore selecting artifacts from a product and through GI graft annotations that capture the variability present, for the selected artifacts, in all the products of a SPL. We intend to use information retrieval metrics as done in our previous work [35], and perform a comparison with ECCO's traceability extraction algorithm.

- *Adapting GenProg ideas into the ECCO approach.* We want to explore how the automated repair work proposed by GenProg [16] can be adapted to the context of SPLs, in particular to solve repair scenarios such as the one illustrated in Section 2. Doing so would require to frame and assess the plastic surgery hypothesis (see [4]) within the realm of SPLs, and integrate the ECCO tool with ASTOR [37], the Java based implementation of GenProg principles.

We hope that the overview we provided can help to draw the attention to the great synergies existing between GI and SPLs, and entice the readers to pursue and tackle the challenges identified.

Acknowledgments

5. REFERENCES

[1] W. K. G. Assunção. Search-based migration of model variants to software product line architectures. In *Doctoral Symposium, International Conference on Software Engineering (ICSE)*, 2015. to appear.

[2] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. Extracting variability-safe feature models from source code dependencies in system variants. In *Genetic and Evolutionary Computation Conference, GECCO*, 2015. to appear.

[3] W. K. G. Assunção and S. R. Vergilio. Feature location for software product line migration: a mapping study. In *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 52–59, 2014.

[4] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In S. Cheung, A. Orso, and M. D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 306–317. ACM, 2014.

[5] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9):577–591, 2007.

[7] G. Botterweck and A. Pleuss. Evolution of software product lines. In T. Mens, A. Serebrenik, and A. Cleve, editors, *Evolving Software Systems*, pages 265–295. Springer, 2014.

[8] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423, 2011.

[9] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. de Almeida. On strategies for testing software product lines: A systematic literature review. *Information & Software Technology*, 56(10):1183–1199, 2014.

[10] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In A. Cleve, F. Ricca, and M. Cerioli, editors, *CSMR*, pages 25–34. IEEE Computer Society, 2013.

[11] E. Engström and P. Runeson. Software product line testing - A systematic mapping study. *Information & Software Technology*, 53(1):2–13, 2011.

[12] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 391–400, 2014.

[13] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. The ecco tool: Extraction and composition for clone-and-own. In *International Conference on Software Engineering (ICSE)*, 2015.

[14] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems - A systematic literature review. *IEEE Trans. Software Eng.*, 40(3):282–306, 2014.

[15] C. L. Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[16] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[17] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: a survey and directions for future work. In S. Gnesi, A. Fantechi, P. Heymans, J. Rubin, K. Czarnecki, and D. Dhungana, editors, *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 5–18. ACM, 2014.

[18] M. Harman, Y. Jia, and W. B. Langdon. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, pages 247–252, 2014.

[19] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In M. Goedicke, T. Menzies, and M. Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 1–14. ACM, 2012.

[20] M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for reverse engineering. In R. Lämmel, R. Oliveto, and R. Robbes, editors, *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 1–10. IEEE, 2013.

[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. H. P. Kim, C. Kästner, and D. S. Batory. On the modularity of feature interactions. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 23–34. ACM, 2008.

[23] M. A. Laguna and Y. Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Program.*, 78(8):1010–1034, 2013.

[24] W. B. Langdon and M. Harman. Optimizing existing software with genetic programming. *IEEE Trans. Evolutionary Computation*, 19(1):118–135, 2015.

[25] L. Linsbauer, F. Angerer, P. Grünbacher, D. Lettner, H. Prähofer, R. E. Lopez-Herrejon, and A. Egyed. Recovering feature-to-code mappings in mixed-variability software systems. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 426–430, 2014.

[26] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. An incremental and product centric approach to product line evolution. In *Submitted for publication*, 2015.

[27] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In *SPLC*, pages 131–140, 2013.

[28] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Feature model synthesis with genetic programming. In C. L. Goues and S. Yoo, editors, *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, volume 8636 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2014.

[29] R. E. Lopez-Herrejon and A. Egyed. Detecting inconsistencies in multi-view models with variability. In T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors, *ECMFA*, volume 6138 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2010.

[30] R. E. Lopez-Herrejon and A. Egyed. Towards interactive visualization support for pairwise testing software product lines. In A. Telea, A. Kerren, and A. Marcus, editors, *VISSOFT*, pages 1–4. IEEE, 2013.

[31] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. *Computational Intelligence and Quantitative Software Engineering*, chapter Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges. Springer, 2015. Accepted for publication.

[32] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. In *4th International Workshop on Combinatorial Testing (IWCT 2015)*, 2015. to appear.

[33] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In G. Fraser and J. T. de Souza, editors, *SSBSE*, volume 7515 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2012.

[34] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information & Software Technology*, 61:33–51, 2015.

[35] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353–369, 2015.

[36] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, and Y. L. Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In H. A. Sahraoui, A. Zaidman, and B. Sharif, editors, *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014*, pages 50–59. IEEE, 2014.

[37] M. Martinez and M. Monperrus. ASTOR: evolutionary automatic software repair for java. *CoRR*, abs/1410.6651, 2014.

[38] M. Noorian, E. Bagheri, and W. Du. Non-functional properties in software product lines: A taxonomy for classification. In *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE'2012), Hotel Sofitel, Redwood City, San Francisco Bay, USA July 1-3, 2012*, pages 663–667. Knowledge Systems Institute Graduate School, 2012.

[39] G. G. Pascual, R. E. Lopez-Herrejon, M. Pinto, L. Fuentes, and A. Egyed. Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. *Journal of Systems and Software*, 103:392–411, 2015.

[40] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. García-Sánchez, J. J. Merelo, V. M. R. Santos, and K. Sim, editors, *Genetic Programming - 17th European Conference, EuroGP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149. Springer, 2014.

[41] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu, 2008.

[42] D. Rattan, R. K. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information & Software Technology*, 55(7):1165–1199, 2013.

[43] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[44] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: a case study in software product lines. In *Proceedings of ICSE*, pages 492–501, 2013.

[45] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Software Technology*, 55(3):491–507, 2013.

[46] F. J. van d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer, 2007.

[47] M. O. Ward, G. G. Grinstein, and D. A. Keim. *Interactive Data Visualization - Foundations, Techniques, and Applications.* A K Peters, 2010.

[48] P. Zave. Faq sheet on feature interaction. http://www.research.att.com/ pamela/faq.html.