Assessing Product Line Derivation Operators Applied to Java Source Code: An Empirical Study

João Bosco Ferreira Filho, Simon Allier, Olivier Barais, Mathieu Acher, Benoit Baudry

Assessing Product Line Derivation Operators Applied to Java Source Code: An Empirical Study

João Bosco Ferreira Filho, Simon Allier, Olivier Barais, Mathieu Acher
INRIA and IRISA
Université Rennes 1, France

Benoit Baudry
INRIA and SIMULA RESEARCH LAB
Rennes, France and Lysaker, Norway

ABSTRACT

Product Derivation is a key activity in Software Product Line Engineering. During this process, derivation operators modify or create core assets (e.g., model elements, source code instructions, components) by adding, removing or substituting them according to a given configuration. The result is a derived product that generally needs to conform to a programming or modeling language. Some operators lead to invalid products when applied to certain assets, some others do not; knowing this in advance can help to better use them, however this is challenging, specially if we consider assets expressed in extensive and complex languages such as Java. In this paper, we empirically answer the following question: which product line operators, applied to which program elements, can synthesize variants of programs that are incorrect, correct or perhaps even conforming to test suites? We implement source code transformations, based on the derivation operators of the Common Variability Language. We automatically synthesize more than 370,000 program variants from a set of 8 real large Java projects (up to 85,000 lines of code), obtaining an extensive panorama of the sanity of the operations.

1. INTRODUCTION

Many domains are concerned with some form of variation in software artifacts—from managing changes to software over time to supporting families of related products (i.e., *software product lines*—SPL). Numerous theories, techniques, tools and languages have been developed to indicate which parts of a system vary, and how a particular variant (product) is produced [3, 4, 20, 24, 30].

SPL developers typically add, remove or replace program elements, like classes, methods or statements, to vary their functionality [3] or their non functional properties [11, 26]. *Add, remove and replace* are **operators**, which applied to program elements gives us **operations** (or transformations, if we use the vocabulary of source code manipulation). For instance, the directives of the C preprocessor (#if, #else, #elif, etc.) can be used to conditionally include parts of files and activate or deactivate a portion of code at compile time [15,17,19]. At the modeling level, the same principle can be applied: presence conditions are specified over different kinds of elements (e.g., classes, associations, attributes in class diagrams) [13, 23].

Given a programming or modeling language, numerous language constructs can be varied, changing the structure or behavior of a program or a model. In the case of Java, developers can add a new class, remove a statement, substitute a field or many other variational operations. Thus, developers are confronted to understand *what* can vary in a software artifact while mastering the technical means of *how* to realize the variability.

In particular, not all constructs of a language are subject to variation because of the numerous well-formedness and domain-specific rules. For instance, removing a return statement in a non-void Java method yields compilation errors; adding a call to an external public method without an import to its respective class either; replacing the type of a parameter by another unrelated type is unlike to produce a correct variant, etc. Some operations will (obviously or intricately) lead to syntactic or semantic errors; some others not. Additionally, a transformation may yield a valid program in one context and an invalid one in another (e.g., removing a class can break a program, if it is being used by another class).

Although generic foundations and tools for correctly managing variations are emerging [4, 8], it is still a very hard task [9]. Some of them concentrate on refactoring the system that is subject to variation [1, 18], some others require specific target formalisms [6]; yet, in many cases, variability management needs to be seamless and non-intrusive, as companies do not want to afford changing their software or processes and tools to cope with variability. One possible solution is to use an orthogonal variability language like CVL (Common Variability Language)[1] [12]. Using CVL, one can express, regardless of the target language, variational information that can be further used to generate variants of a model. This can provide a non-intrusive mechanism to handle variability in existing (legacy) software.

What if we could *correctly* use CVL to *directly* vary Java programs? We would be able to express and manage variability information about any peace of code orthogonally and without changing the target program. We could consider to vary not only coarse-grained elements, like components or entire modules, but also fine-grained constructs, like instructions inside of a method. These fine-grained operations could be then composed into multiple operations to vary a set of instructions corresponding to a systems' feature. Such abilities serve beyond SPL engineering and could be helpful to any kind of program variation approach (e.g., mutant-based testing [2], approximate computation [32], program sketching [27]). The unfortunate intuition is that not all CVL-based operators are relevant and directly applicable, e.g., some of the derivation operators will lead to Java program variants that do not compile. The first goal of this paper is to empirically verify this hypothesis. The second goal is to characterize the effects of each CVL-based operator applied to programs: which operators are directly applicable? which

[1]http://www.omgwiki.org/variability

operators need to be specialized? Our work seeks to assess the applicability of the CVL derivation operators with respect to existing Java code elements, as well as quantifying how prone each kind of derivation operator and Java construct are to generate valid or invalid programs. This understanding may help to validate, refine, or disprove what kinds of variations on program elements the variability-based tools (e.g., IDE, refactoring) or paradigms (e.g., feature-oriented programming) should support.

We conduct an empirical study in which we exhaustively apply derivation operators to random source code elements in a set of 8 real large Java programs with up to 85,000 lines of code. For each operation, we verify if the resulting Java variant compiles and if the test suite passes. Statistical data and synthesized variants are then used to characterize which language constructs are likely to vary or require specific transformations. Specifically, we answer to the following research questions:

(1) Which CVL-based operations and operators are more likely to generate invalid and valid programs?

(2) Which program elements are more likely to vary without breaking the program? Which ones are not?

(3) Can we, and how could we identify operators to be specialized?

We measure, report, and discuss the percentages of valid and invalid generated programs (among 370,000 variants) providing a quantified panorama of the results. From a qualitative perspective, we review and analyze the resulting Java variants with the help of dedicated tools.

The remainder of the paper is organized as follows. Section 2 gives background information about CVL and the motivations of the study. Section 3, introduces our CVL-based approach to automatically synthesize Java variants. Section 4 defines in detail our experiment. Section 5 analyses the results and discuss them. Section 6 presents the threats to validity of the experiment. Section 7 discusses related work. Section 8 concludes the paper and presents future work.

2. BACKGROUND AND MOTIVATIONS

2.1 The Common Variability Language

We have chosen to use CVL because it encompasses many SPL approaches and because it fulfills the requirements of our long term view: *to orthogonally express, reason and realize variability information of a system but neither changing it nor introducing new language constructs to cope with variability.* CVL is an effort from various partners from both academia and industry to standardize variability in model-based product line. Using CVL, one can specify and resolve variability over any artifact that could be abstracted as a model.

An SPL defined with CVL has three main parts: the Variability Abstraction Model (VAM), which is the tree-based structure (equivalent to feature models, see VAM in Figure 1, in which VSs are *Variability Specifications*, analogue to features), the base model (the core assets; in Figure 1, it is the existing Java code), and a Variability Realization Model (the mapping between both, see VRM in Figure 1). These mapping relationships can have different semantics and we name them **realization/derivation operators** (in the CVL specification, they are called *variation points*).

In this paper, we are interested on studying the effects of applying these realization operators to Java programs, in any kind of program construct. We concentrate on the last two parts af an SPL defined with CVL: the mapping operators and the base model elements. As for the VAM, we will consider only boolean features and we abstract about its configuration: the goal is to study what happens when, individually, a VRM operator is triggered by a selection or deselection of a feature, being this study independent from the VAM part.

CVL Derivation Operators

Following, we list each of the CVL realization operators considered in this work, explaining how they are implemented in the Java context and briefly exemplifying them.

- Object Existence. It is an operator that expresses whether a determined object will make part or not of the derived variant; its execution implies on deleting or adding any source code element (e.g., statements, assignments, blocks, literals, etc.) from the original program.
- Link Existence. It expresses whether there is a relationship or not between two elements, in the case of Java programs, we consider as a link any relationship between classes: association, composition, inheritance, etc. The execution of this variation point implies on removing or adding statements that refer to the associated class (e.g., to remove an *extends Class A* from a class' header).
- Object Substitution. It expresses that a determined program element will be replaced by another of its same type, e.g., a *method* substituted by another *method*.
- Link End Substitution. It expresses that a relationship between a class A and a class B will be replaced by another relationship of the same type between class A and a third class C (e.g., A extends C instead of A extends B).

2.2 Motivations

Our main motivation relies on the use of CVL to vary Java programs. As illustrated in Figure 1, CVL has a derivation engine that executes the operational semantics of its operators (e.g., Object Existence) when applied to a target element (e.g., statements in an existing Java code). The derivation process generates products (programs in our case) that may or may not be correct (e.g., uncompilable Java code or programs with run time errors). Determining whether an operator applied to a program construct is going to yield valid or invalid programs can be hard. In the case of Java, besides depending on the own semantics of the CVL operator, the success of a product line operation may also depend on: the type of the program construct, the other constructs to which it is connected, how it is being used in the current situation, etc. Manually evaluating all these cases can be endless.

Therefore, our work seeks to determine the "safety" of these operations automatically. Figure 1 shows percentages over different operations; they reveal how likely is an operator to work when applied to a given program construct. **These percentages are the output of our experiment** and may be useful for different applications. Two possible applications are to: make recommendations about which operations to use, and specialize derivation operators that do not work as they are. The ultimate goal is to make the use of CVL safer, so that the chances of having a bad CVL design are lower. For example, Figure 1 illustrates that removing a field *name* or the entire class *Client* has less than 10% of chances of resulting in a valid program after derivation, while removing the annotation element can be harmless.

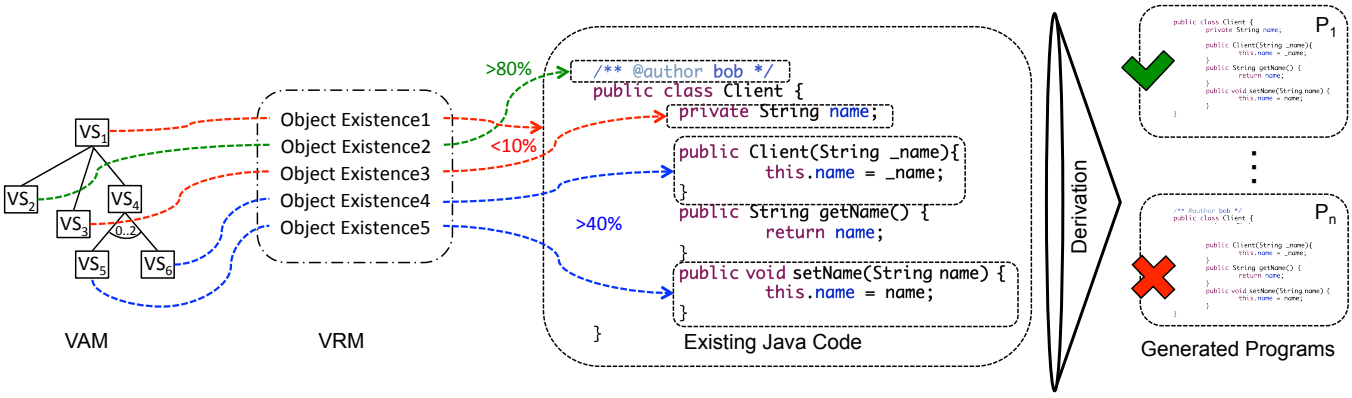

Figure 1: Directly using CVL with Java code.

3. AUTOMATIC SYNTHESIS OF JAVA PROGRAMS WITH CVL

This section presents an overview of the approach to automatically synthesize Java programs using CVL. This automation is the foundation to perform the empirical assessment.

3.1 Definition

In CVL, the operators are always linked to a target element. Consequently, we will further refer to the definition of operation type T as a pair $\langle O, E \rangle$, in which O is a type of CVL operator (from the aforementioned list) and E is a type of targeted program element (e.g., code statement, class, package). Therefore, an operation $op = \langle o, e \rangle$ of type T is an operation in which $o \in O$ and $e \in E$. With a derivation engine δ, we can apply op in a given Java program P, having $\delta(op, P) = P'$, where P' is a generated program.

Given an operation op, a program P that successfully compiles and a test suite TS that passes on P, the possible results for a transformed program $P' = \delta(op, P)$ are:

1. P' is syntactically incorrect and contains compilation errors–P' is a *counterexample*;
2. P' is syntactically correct and successfully compiles but at least one test case in TS fails–P' is a *variant*;
3. P' compiles and all test cases in TS passes–P' is a *sosie*[2].

3.2 Process overview

Figure 2 shows an overview of the process of generating a program P'. First, we extract the Abstract Syntax Tree (AST) of P, which provides the set of program elements and their relationships. This step makes possible to handle P as a model, therefore we can use the concept of model-based SPL with CVL. Second, we use the AST of P and the list of CVL realization operators as input to the derivation. In the derivation step, we pick a random program element and a random operator, composing an operation and then applying it by using the CVL derivation engine; the result of this is a generated AST'. As a third step, we print back as source code the AST', having as result a generated program P'. Finally, we try to compile P' and also to test it against the test suite, which evaluates P' as a counterexample or a variant or a sosie.

[2]Sosie is a French noun that means "look alike" and it has been previously defined in [5].

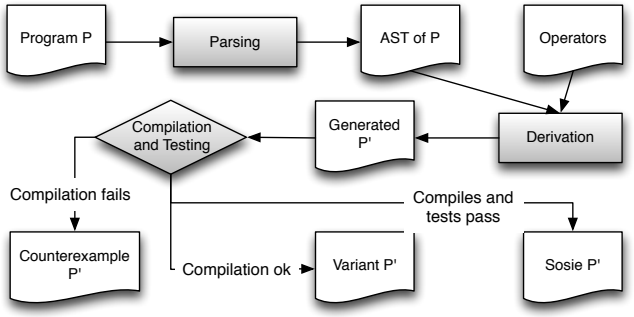

Figure 2: Process to generate variants of Java programs.

Examples of generated programs

Following, we exemplify the three possible evaluations of a program generated by our approach—*conterexample, variant or sosie*—using real examples from an existing Java program; we exemplify that the same CVL-operator (e.g., *Object Substitution*) can yield both valid and invalid programs. Listing 1 shows an excerpt of a generated program that does not compile; precisely, it is the result of an *Object Substitution* in the statement of the line 38 inside the constructor of the class *UniformReservoir* of the *metrics*[3] project. The replaced statement is commented (instead of actually deleted, in order to facilitate visualization and retrieval) and a new statement is placed right after. In this case, one of the reasons it does not compile is because the variable *registry* was not declared before.

Listing 1: Object Substitution generating a counterexample.

```
//class com.codahale.metrics.UniformReservoir, line 38
public UniformReservoir(int size) {
        this.values = new AtomicLongArray(size);
        for (int i = 0; i < values.length(); i++) {
            values.set(i, 0);
        }
        //subtitution
        //count.set(0);
        registry.register(name(prefix, "mean-get-time"));
             }
```

[3]http://metrics.codahale.com/

Differently, we have cases in which a substitution of a program element does not imply any error. Listing 2 shows one of these cases and, like in the previous operation, a statement is replaced by another. In this case, the replaced statement is an independent method call, as well as the inserted one, which is a static method call. However, the transformed program does not have the same behaviour of the original one, therefore it does not pass on the test suite of the original one. The reason is because the replaced statement plays a role on the functionality of the *time* method. This program is a variant.

Listing 2: Object Substitution generating a variant.

```
//class com.codahale.metrics.Timer, line 101
public <T> T time(Callable<T> event) throws Exception {
       final long startTime = clock.getTick();
       try {
           return event.call();
       } finally {
           //substitution
           //update(clock.getTick() - startTime);
           com.codahale.metrics.ThreadLocalRandom.
           current().nextLong(); }}
```

In some situations, an operation can generate compilable variants and also preserve the behaviour of the original program– *sosie*. Following, Listing 3 presents a sosie generated from a replacement of a literal value by another of the same type (the *string"csv-reporter"* is replaced by *"m5_rate"*), therefore not leading to compilation errors; besides, this literal did not play an important role on the program execution and its behaviour remained unchanged.

Listing 3: Object Substitution generating a sosie.

```
//class com.codahale.metrics.CsvReporter, line 135
private CsvReporter(MetricRegistry registry,
               File directory, Locale locale,
               TimeUnit rateUnit, TimeUnit durationUnit,
               Clock clock,
               MetricFilter filter) {
       //substitution
       super(registry,/**Type:Literal"csv-reporter"**/
       "m5_rate", filter, rateUnit, durationUnit);
       this.directory = directory;
       this.locale = locale;
       this.clock = clock; }
```

These examples also apply to the other types of CVL operators. We could see that the success of an operation depends on the kind of the targeted element. It is expected that it would not be possible to modify or remove some program elements without leading to compilation errors (e.g., remove a return keyword from a method with non-void value type). On the other hand, we can easily expect that some statements that do not have any impact on the program execution, like log statements, could be removed without any further problem.

4. EXPERIMENT

In this Section, we present in details the empirical study we conducted in order to assess the product line derivation operators in the context of Java programs.

4.1 Goal and Research Questions

The main objective of the experiment is to assess the safety of CVL operators when varying Java programs. We specifically aim at answering the following research questions:

RQ1. Which CVL-based operations (pairs of operators and program elements) are more likely to generate invalid and valid programs? Answering RQ1 can give us insight about transformations that will always lead to counterexamples or transformations that will always lead to variants or sosies. We will also be able to quantify the chances that each operation has to yield valid or invalid programs.

We also want to investigate the proportion of valid and invalid programs individually, for both program elements and derivation operators, which gives us RQ2a and RQ2b.

RQ2a. Which program elements are more likely to vary without breaking the program and which ones are not? We will be able to know, for example, if removing program elements that are blocks of code is more likely to generate correct programs than if removing single instructions.

RQ2b. Which CVL operators are more likely to result in correct programs? Knowing, for example, if Object Existence performs better than Object Substitution.

RQ3. Can we, and how could we identify operators to be specialized? Qualitatively analysing the results can help to design better operators.

4.2 Measurement Methodology

We measure the percentage of non-compilable, compilable program variants and sosies generated by a given operator applied to different program elements. We want to observe these percentages with respect to the operators, the program elements and the pairs operator & program element (operations). For each analysed program, our experimentation algorithm performs one transformation per time and tries to compile the transformed program, if it compiles, we proceed to run the test suite, checking whether it passes or not.

4.3 Experiment Variables

We define our variables according to the theory of scales of measurements; additionally, they are also classified as independent, dependent or controlled variables. Independent and controlled variables influence dependent variables, but the controlled ones remain unchanged during the entire experimentation. Table 1 presents the experiment variables with their classification and the range of values they can assume during the experiment. The number of non-compilable, compilable and programs with preserved behaviour is dependent on the operator and the program element[4] used in the program transformation. We perform the experiment for a controlled set of 8 input programs.

4.4 Subject Programs

The data set of our experiment is composed by 8 widely-used open source projects. The first selection criterion is that the project is structured to be handled by Maven: a tool for building and managing Java projects. The second one is that they need to have a good test suite (a statement coverage greater than 70%); they are all expressed in JUnit. Table 2 shows the included projects and some relevant properties for the experiment.

The size of each program ranges from 1 to 80 KLOC and the number of classes from 23 to 803; they are in the category

[4]The complete list of program elements can be found in the Spoon API: http://spoon.gforge.inria.fr/mvnsites/spoon-core/apidocs/index.html

Table 1: Experiment variables.

Name	Abbreviation	Type	Scale Type	Unit	Range
CVL Realization Operator	operator	Independent	Nominal	Text	{ObjectExistence, LinkExistence, ObjectSubstitution, LinkEndSubstitution}
Program Element	element	Independent	Nominal	Text	{constructor, class, parameter, statement, etc}
Non-compilable programs	counterexamples	Dependent	Ratio	%	[0,100]
Compilable programs	compile%	Dependent	Ratio	%	[0,100]
Variants with preserved behaviour	sosie%	Dependent	Ratio	%	[0,100]
Original input program	input	Controlled	Nominal	Text	see Table 2

of APIs and frameworks—programs that are used by other programs. All of them have a good test coverage percentage, ranging from 79% to 94%. None of the programs have a compilation time greater than 10 seconds, which helps on the total time for running the experiments. However, their testing time ranges from 7 to 144 seconds[5].

4.5 Protocol

The experiment is designed to randomly explore the possible transformations that can be done in a given program, having its AST nodes and the four operators as the universe to be sampled. Algorithm 1 defines the protocol to run the experiments. It takes as input the program to be transformed and returns the data we use further to analyse the transformations and the AST elements (we get either a counterexample, a variant or a sosie for each random transformation applied). Our stopping criteria is not strict and it is defined by the amount of computational resources available in the Grid5000[6]—we seek to achieve a reasonable statistical relevance.

Data: P, a program to vary
Result: values for the dependent variables of Table 4
O={the four kinds of CVL operators}
E={elements in the AST of P}
while *resources_available* **do**
  randomly select $o \in O$
  randomly select $e \in E$
  $op \leftarrow \langle o, e \rangle$
  $P' \leftarrow derive\ (op, P)$
  if *compile(P') = true* **then**
    if *test(P') = true* **then**
      store P' as a sosie
    else
      store P' as a variant
    end
  else
    store P' as a counterexample
  end
end

Algorithm 1: The experimental protocol for creating, applying the operations and evaluating the generated program.

[5]CPU: Intel Xeon Processor W3540 (4 core, 2.93 GHz), RAM: 6GB
[6]www.grid5000.fr

5. ANALYSIS

5.1 Results

Table 3 presents the results after running the experiments for the 8 subject programs, totalizing 196,816 lines of code. We calculate the number of possibilities of applying a specific operator in the universe of the 8 programs (the *candidate* column). The number of candidates for Object Existence is simply the number of nodes in the AST; for the Object Substitution, it is the sum of the squares of the number of elements of each AST type present in the programs; for Link Existence is the number of fields plus the number of inheritance links; and for the Link End Substitution, it is the number of fields squared plus the number o inheritance links squared.

The *Trial* column describes how many times we applied a transformation containing the given operator. Given the number of candidates, the number of trials, and a confidence of 99%, we calculate the margin of error for each operator. This margin holds meaning if one wants to consider the results as probabilities inside our universe. An example of interpretation is: the probability of having a program that compiles after removing a random program element is between 10.97% and 11.97% (compile% = 11.47 and margin of error = 0.50).

Panorama of operations. There are 86 possible combinations between operators and types of program elements in our data set. In Table 4, we show the 15 first and the 15 last operations ordered by their compilation percentage. The complete table can be found in the paper web site. The first column refers to the type of operator: OE (Object Existence), OS (Object Substitution), LE (Link Existence) and LS (Link End Substitution). The second column is the affected program element. We also show the number of possibilities for each transformation and how many times we actually apply them in our experiments.

Panorama of types of program elements. Figure 3 shows the results for 7 groups of program elements, independently from the kind of operator. Each of the vertical bars represent the compilable and sosie percentages for a given project (they are arranged in the same order of Table 2). We measured the dependent variables for all 42 program elements, and the detailed results for each of the projects can be found in the paper's web page. However, to fit the results to be seen in this paper, we have selected 20 elements and grouped them according to their common function. The first group is the *if*, containing the *if* and the *conditional* (i.e., A?B:C) program elements. The *loops* group contains the *do*, *while*, *foreach* and *for* elements. The *invocations* group is composed by the *invocation* (i.e, a method call such as .a() ,

Table 2: Descriptive statistics about our experimental data set

	#LoC	#class	#test case	#assert	coverage	#statement	compile time (sec)	test time (sec)
JUnit	8056	170	721	1535	82%	2914	4.5	14.4
EasyMock	4544	81	617	924	91%	2042	4	7.8
JBehave-core	13173	188	485	1451	89%	4984	5.5	22.9
Metrics	4066	56	214	312	79%	1471	4.7	7.7
commons-collections	23559	285	1121	5397	84%	9893	7.9	22.9
commons-lang	22521	112	2359	13681	94%	11715	6.3	24.6
commons-math	84282	803	3544	9559	92%	47065	9.2	144.2
clojure	36615	150	NA	NA	71%	18533	105.1	185

Table 3: Results for the operators.

	candidate	trial	%trial	margin of error	compile	compile%	sosie	sosie%
Link Existence	11248	7247	64.43	0.70	856	11.81	539	7.44
Link Substitution	14869609	85459	0.57	0.40	3851	4.51	3572	4.18
Object Existence	626258	79913	12.76	0.30	17559	21.97	6994	8.75
Object Substitution	14706362886	203566	<0.01	0.20	18776	9.22	12127	5.96
Total	14721870001	376185	<0.01	0.20	41042	10.91	23232	6.18

where a is a method), *unary operators* and *binary operators*; we see the two last as invocations of methods (e.g., a + b is equivalent to add(a,b)). The *read* group contains the program elements in charge of access: *variable*, *field* and *array access*. The write group is composed by the *assignment* and *operator assignment* program elements. The *new* group contains the elements responsible to create objects or primitive types (the case for literal): *new array*, *new class* and *literal*. In the *exception* group, we gathered the *catch*, *try* and *throw* elements. In Table 5, we show the values for the variance, standard deviation, mean and margin of error for each of the aforementioned group of AST elements.

Table 5: Variance(σ^2), standard deviation(σ), mean(μ) and margin of error (ME) for the compilation% of the 7 groups of Figure 3.

	σ^2	σ	μ	ME
if	47.44	6.89	26.32	0.01
loop	54.90	7.41	34.16	0.02
invocation	13.31	3.65	12.29	0.00
read	2.22	1.49	6.57	0.00
write	41.50	6.44	24.42	0.01
new	111.49	10.56	33.90	0.01
exception	40.81	6.39	16.62	0.02

We excluded from Figure 3 program elements that have never compiled after being affected.

5.2 Visualizing the Results

Due to the large amount of data produced as result of the experiment, we had to provide means to ease the visualization of the transformations. Figure 4 shows the web-based visualization tool we built to achieve this task. First, we provide a global view of the input program by packages (see ①), within each package we have the classes, which are represented as long rectangles with colored lines inside. It is possible to click on those rectangles to zoom in the classes (see ②). Once zoomed, it is possible to see and access each of the colored lines that make part of a class; they represent code locations (a line number) that received transformations. The red portions of the lines represent the amount of transformations in that place of the code that did not succeed to compile, while blue portions represent the ones that compiled and the green portion the ones that resulted on sosies.

Furthermore, we made possible to click on each line to visualize the list of the transformations done in a given place (see ③). This third view provides details on the actual number of transformations performed in that code location, the name of the applied transformation and its status (0 means it compiled and passed the tests, -1 it compiled and -2 it did not compile). In the transformed code, we comment everything that was supposed to be removed/substituted in a transformation in order to let the user compare the before and after the transformation. In the specific case of Figure 4, the first transformation erased the first parameter of a method call and the second one replaced the *"null"* keyword by the *"unchecked"* string. The visualization tool is available in the paper's web site: `http://varyjava.barais.fr/`.

5.3 Discussing the Research Questions

5.3.1 RQ1. Safe and Unsafe Operations

We refer to Table 4 to discuss RQ1; it shows an excerpt of the comprehensive panorama of the operations' success percentages. In its 12 last lines, we can find 12 operations that always lead to counterexamples. Meanwhile, if we observe the 2 first lines of the table we will find 2 operations that have always generated variants or sosies, followed by 72 kinds of operations that worked at least one time. The complete results can be found in the paper webpage. Some elements of a Java program have an optional nature with respect to correctness.

There are many ways of varying a Java program using CVL. Only 14% of operations (12 out of 86) never resulted in valid programs. The other 86% resulted in compilable programs in at least one attempt. The frequencies may be low while some of the operations exhibit high percentages (see also RQ2). A direct application of CVL operations is thus generally not effective, suggesting to specialize some operators (see also RQ3).

Table 4: Global results for the 15 first and the 15 last operations ordered by compilation %.

operator	program element	candidate	trial	%trial	compile	**compile%**	sosie	sosie%
OE	AnnotationType	46	23	50.00	23	100.00	19	82.61
OE	Continue	124	31	25.00	31	100.00	9	29.03
OE	ForEach	888	330	37.16	325	98.48	47	14.24
OS	SuperAccess	60348	189	0.31	186	98.41	183	96.83
OS	ThisAccess	5790636	2282	0.04	2203	96.54	2006	87.91
OE	SuperAccess	456	86	18.86	83	96.51	24	27.91
OE	While	609	95	15.60	88	92.63	18	18.95
OE	For	3461	251	7.25	230	91.63	60	23.90
OE	Break	1008	121	12.00	110	90.91	74	61.16
OE	OperatorAssignment	1825	153	8.38	137	89.54	53	34.64
OE	If	12859	2175	16.91	1851	85.10	587	26.99
OE	Annotation	3802	903	23.75	699	77.41	655	72.54
OE	Throw	3092	523	16.91	370	70.75	150	28.68
OS	Annotation	3150678	1591	0.05	1019	64.05	980	61.60
OE	Synchronized	95	27	28.42	16	59.26	1	3.70
...	...	...	...	...	...	...	...	...
OS	Parameter	172555379	13594	0.01	12	0.09	12	0.09
OE	Method	18906	3998	21.15	3	0.08	3	0.08
OE	Parameter	28701	4494	15.66	2	0.04	1	0.02
OE	Catch	602	218	36.21	0	0.00	0	0.00
OE	Class	2477	309	12.47	0	0.00	0	0.00
OE	Enum	61	4	6.56	0	0.00	0	0.00
OE	Interface	671	52	7.75	0	0.00	0	0.00
OS	Break	409674	192	0.05	0	0.00	0	0.00
OS	Case	2035772	530	0.03	0	0.00	0	0.00
OS	Catch	51158	650	1.27	0	0.00	0	0.00
OS	Continue	4554	27	0.59	0	0.00	0	0.00
OS	Do	916	8	0.87	0	0.00	0	0.00
OS	Field	13019255	3639	0.03	0	0.00	0	0.00
OS	LocalVariable	162483228	6604	0.00	0	0.00	0	0.00
OS	Throw	2361268	1270	0.05	0	0.00	0	0.00

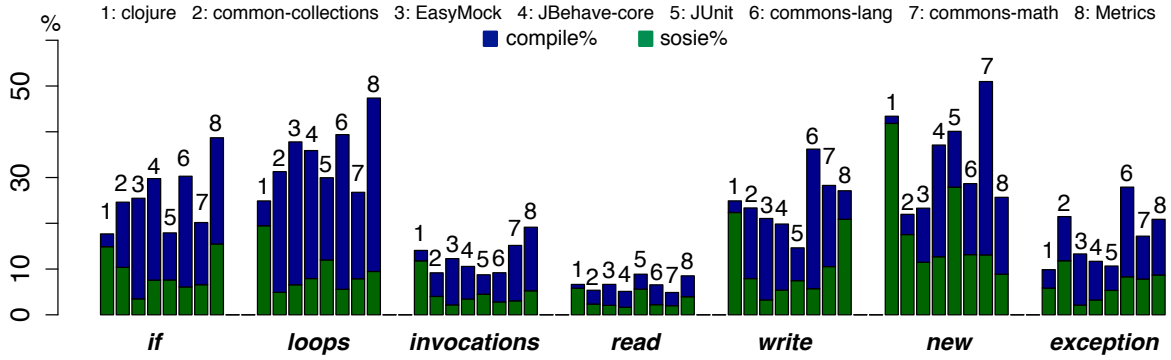

Figure 3: Results by categories of program elements for the 8 projects.

For example, if we remove a *"continue"* from a loop it will continue syntactically correct, and maybe even semantically, since a *"continue"* can be used for optimization purposes, not implicating changes in the loop semantics. On the other hand, there are some Java constructs that can be considered as mandatory with respect to others and therefore must be handled carefully.

As CVL is designed to be generic to any target language, the fact that it is easier to design unsafe CVL models with respect to Java is acceptable and even expected [9]; it is unfeasible to anticipate every possible domain-specific syntax or semantics rules for any language; therefore an specialization step is strongly advised, and it is further addressed in the RQ3 discussion. Despite of having 12 transformations that never yield valid programs, we observe that they are the minority: 14% (12 out of 86). This indicates that there are several possibilities for varying a Java program without crashing it.

5.3.2 RQ2. Safety of Program Elements/Operators

In order to address RQ2a, we can pick the operations that have Object Existence as their operator and blocks of code as elements: *Do, For, ForEach, While, If, Throw.* We can observe in Table 4 that their variant percentage are from 70% (in the case of the *Throw*) to 98% (in the case of the *ForEach*). We can also observe from Table 5.1 and Figure 3 that the mean for *loops*, which are blocks of code, is the greatest comparing to the others program elements.

> *We confirm the idea that blocks of code are easier to vary; their success percentages are the highest (from 70% to 98% of derived programs are valid). Derivation operators over Do, For, ForEach, While, If, Throw are applicable for realizing fine-grained variability (typically inside methods).*

We refer to Table 3 to discuss RQ2b. We can see that only about 9% of the transformations based on Object Substitution have succeed to generate correct programs, while the

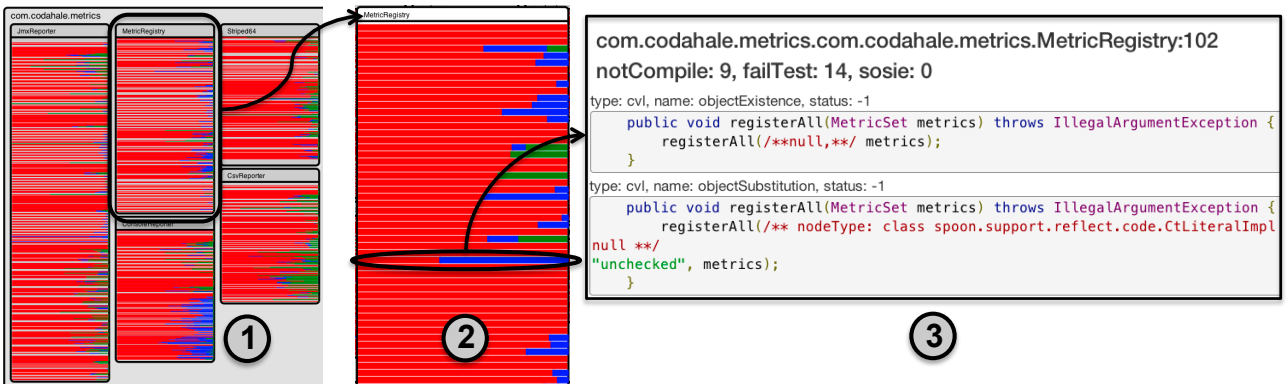

Figure 4: Visualizing the transformations.

ones based on Object Existence had about 21% of success. This is due to the fact that Object Existence is translated into fewer operations than Object Substitution, which encompasses removing one element (OE) and adding a substitute. An interesting number is the one of Link Existence, implying that, in some situations, we could remove associations between Classes and the generated program still compiled and even passed the test suite.

5.3.3 RQ3. Identifying Operators to be Specialized

To address RQ3, we can refer to the extensive results of percentages and to our visualization tool. We took the role of the user to investigate some of the operations in one of the data set projects. We could learn from this experience that there is a straightforward path to follow. (1) We looked into most popular types of program elements first (e.g., Class) and checked how the operations performed. Two possible outcomes are: the operation has a high success percentage or it has a low one; if it has a high one we can keep the operation.

> *If an operation has a low compilation%, it does not necessarily mean that the operation is irrelevant. Our qualitative review rather suggests that there is room for specializing operators through simple adaptations or complex refactoring (e.g., based on static analysis).*

Determining when an operation is irrelevant and not useful needs further investigation and typically requires knowledge about the host language (here Java) and the purpose of the variation to the user. The problems induced by some derivation operators can also justify the emergence of paradigms (e.g., feature-oriented programming) that try to provide coarse-grained operators for variability.

(2) We analyse the code, with the help of our tool and we check why the operation performed poorly. (3) We assess if the errors occasioned by the operations are recurrent and can be fixed systematically; in case it is true, we can do a simple adaptation of the operator or a static analysis based one. One example of simple adaptation is with a *try-catch*. We know that a *"try"* block is often followed by a *"catch"* or a *finally* one, therefore removing a catch have great chances of giving compilation errors, to avoid this, we can simply condition the removal of a catch to the removal of its corresponding *"try"*.

On the other hand, some specializations need the assistance of static analysis. For example, we would expect to be able to remove a Class using CVL, however this operation has a compile% close to 0; we could design a *ClassExistence* operator that would remove a given Class and all the other Classes/Interfaces to which it points. The same happens with replacing a *"field"* by another. Instead of having an Object Existence to a *"field"*, we would have a specialized operator called *Field Existence*, which would be responsible by not only the existence of the given field, but also by any occurrence of that field in the program.

6. THREATS TO VALIDITY

Our experiment has the necessary conditions for *causality*. No other changes are done in the programs by each iteration, we only perform one operation at a time (see Algorithm 1), therefore the changes in our dependent variables are only related to the execution of the given operation.

Internal Validity: One potential threat for *internal validity* is the number of trials with respect to the possible number of operations in our universe. We have addressed this threat by controlling the margin of error for the operations, having it always less than 1% and using a confidence level of 99%. However, some program elements were not numerous enough for being representative, such as *annotation types*; they are rarely used and therefore our experiments may not be conclusive for these program elements.

External Validity: Regarding the representativeness of the transformations, in total, we generated, compiled and tested 376,185 programs. We tried to compute the maximum number of operations as possible, given the available resources; if we multiply the total number of transformations by their compilation time, added to the testing time (when compilation is ok) of the 8 projects, we have a total of around 97 days of computation in a personal computer.

By calculating the variance and standard deviation of transforming specific program elements over our 8 different subject programs, we could notice some discrepancy among them. This fact can be an evidence that factors such as choices of design and programming style, can have an influence on the compilation percentages; it needs to be further explored.

Approach Generality: We chose Java as it is a widely used programming language, however our experiments can be reproduced to analyse other languages. The essential steps are: (1) Perform executions of the operations over the constructs of the chosen language in a set of examples; (2) Collect and categorize the generated variants in succeeded and not succeeded, together with the used transformations; (3) Quanti-

tatively analyse the best and the worst transformations with respect to a criteria; (4) Qualitatively analyse subsets of operations + transformed programs by domain expert and specialize operators.

7. RELATED WORK

Our work is at the crossroads of research related to (1) variability and software product lines, (2) Java, and (3) program synthesis and transformation.

Variability Approaches. Different approaches to represent variation have been proposed. For instance, the directives of the C preprocessor (#if, #else, #elif, etc.) conditionally include parts of files. They can be used to activate or deactivate a portion of code [15, 17]. Superimposition is a generic composition mechanism to produce new variants, being programs (written in C, C++, C#, Haskell, Java, etc.), HTML pages, Makefiles, or UML models. Software artefacts are composed through the merging of their corresponding substructures [4]. Voelter and Groher *et al.* illustrated how negative (i.e., annotative) and *positive* (i.e., compositional) variability [31] can be combined. *Delta modeling* [25] promotes a modular approach to develop SPLs. The deltas are defined in separate models and a core model is transformed into a new variant by applying a set of deltas. At the foundation level, the *Choice Calculus* [8] provides a theoretical framework for representing variations (being annotative or compositional).

The Common Variability Language (CVL) has emerged to provide a solution for managing variability in any domain-specific modeling languages [12, 28]. CVL provides both the means to support annotative, compositional, or transformational mechanisms. CVL thus shares similarities with other variability approaches. The effort involves academic and industry partners and pursues the goal of providing a generic yet extensible solution. Our work provides further empirical results of CVL as well as a tooling infrastructure capable of operating over Java programs.

Empirical Studies and Variability. Empirical studies have been conducted to further understand variability-intensive systems (e.g., [1,15,17,19,21]). For instance, Liebieg *et al.* [17, 19] analyzed annotations of the C preprocessor over 40 open sources projects; metrics and qualitative assessment of the discipline of annotations are reported.

Most of the empirical studies focus on studying the existing practices of variability, trying to understand how developers implement configurable systems or SPLs. These studies provide insights for tooling builders, designers of languages, or developers. Typically new tools or programming paradigms emerge to better support the activities of practitioners [14, 16, 22]. Our approach differs in that we explore all possible types of variability transformations a developer could apply in a language, not limiting to a subset of constructs. It is most likely that we explore variability mechanisms that have not been observed in existing projects.

Empirical Studies and Java. Empirical studies have been conducted to understand how developers use Java. For instance, Dyer *et al.* [7] analyzed 18 billion AST nodes to find uses of new Java language features over time. Tempero *et al.* [29] studied the use of inheritance in Java projects. Our empirical study focuses on the possible product line based operations developers can use for deriving Java variants.

Program Synthesis and Transformations. Several authors have developed techniques to synthesize or transform programs [2, 10]. The objectives and applications are multi-fold. For instance, mutation techniques aim simulating faults in order to improve the fault detection power of test suites [2]. The use of product line derivation operators can be considered in this context. We leave it as future work.

We refer to [5] for a thorough discussion about *sosies*, diversity, and existing synthesis techniques. We chose to consider *sosies* for (1) evaluating the potential of CVL transformations in the quest of diversification; (2) challenging further CVL transformations – it is more difficult to synthesize a sosie than just a compilable Java variant.

8. CONCLUSIONS AND FUTURE WORK

This paper presented the first empirical study seeking to systematically understand the effects of a direct application of different product line operations over any Java program constructs. We described a fully automated procedure based on the Common Variability Language (CVL) to synthesize variants of Java programs. Our approach can be conceptually extended to any (modeling) language subject to variation. We implemented the approach and developed an infrastructure capable of performing large-scale experiments with CVL and Java. We performed a substantial empirical study of the impact of variational transformation across a set of Java applications, obtaining 376,185 variants of Java programs. Our results provide quantified knowledge about the safety of CVL operators and the brittleness of the different kinds of Java program elements. The data set, empirical results, and the set of visualisation tools are available online at `http://varyjava.barais.fr/`.

We studied 86 different ways of varying a Java program, out of which many had never been investigated by existing research on variability realization. We give quantitatively-supported insights about product line operators:

- 14% of operations never resulted in valid programs; the other 86% resulted in compilable programs in at least one attempt, but the frequencies are generally low;
- derivation operators (Do, For, ForEach, While, If, Throw) for realizing fine-grained variability (typically inside methods) have highest percentages (from 70% to 98% of derived programs are valid);
- a low compilation% does not necessarily mean that the operation is irrelevant. Our qualitative review rather suggests that there is room for specializing operators through simple adaptations or complex refactoring (e.g., based on static analysis).

Future Work. An immediate research direction is to investigate how to validate, disprove, and specialize derivation operators as part of product line tools (e.g., IDE, refactoring). Our empirical results also suggest to study the design of coarse-grained derivation operators for manipulating variability at a higher level (e.g., beyond individual statements).

Our long-term view is to apply CVL to any language in a safer way. The empirical study opens avenues for further investigations and developments. In the short-term, we plan to implement specialized CVL operators for Java programs, and then running the same experiments with the new set of operators, evaluating how did the refinement helped with the task of generating safer CVL designs; we would also catch less obvious situations that lead to invalid programs and then specialize the operators once again to correct more intricate errors.

Another direction is to investigate the derivation operators in other areas than product line engineering. We plan to apply them to any kind of program variation approach (e.g., mutant-based testing [2], approximate computation [32], program sketching [27]).

9. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *Proc. of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 243–254, New York, NY, USA, 2009. ACM.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[3] S. Apel, D. Batory, C. KÃd'stner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[4] S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[5] B. Baudry, S. Allier, and M. Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA'14)*. ACM, 2014.

[6] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[7] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proc. of the 36th International Conference on Software Engineering*, ICSE'14, 2014.

[8] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, Dec. 2011.

[9] J. B. F. Filho, O. Barais, M. Acher, B. Baudry, and J. Le Noir. Generating counterexamples of model-based software product lines: an exploratory study. In *Proc. of the 17th International Software Product Line Conference*, SPLC '13, pages 72–81, New York, NY, USA, 2013. ACM.

[10] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72. IEEE, 1997.

[11] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE'13*, pages 301–311, 2013.

[12] O. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *Proc. of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.

[13] F. Heidenreich, P. Sanchez, J. Santos, S. Zschaler, M. Alferez, J. Araujo, L. Fuentes, U. K. amd Ana Moreira, and A. Rashid. Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *TAOSD VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114, 2010.

[14] C. Kästner and S. Apel. Virtual separation of concerns – a second chance for preprocessors, 9 2009. Refereed Column.

[15] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE '08: Proc. of the 30th international conference on Software engineering*, pages 311–320, New York, NY, USA, 2008. ACM.

[16] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):Article 14, 2012.

[17] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM.

[18] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proc. of the 2015 International Conference on Software Engineering*, ICSE '15, 2015.

[19] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proc. of the 10th International Conference on Aspect-oriented Software Development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM.

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

[21] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information & Software Technology*, 52(3):324–346, 2010.

[22] M. Ribeiro, P. Borba, and C. Kästner. Feature maintenance with emergent interfaces. In *Proc. of the 36th International Conference on Software Engineering (ICSE)*, 6 2014.

[23] R. Salay, M. Famelis, J. Rubin, A. D. Sandro, and M. Chechik. Lifting model transformations to product lines. In *ICSE'14*, 2014. to appear.

[24] 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 D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE*, pages 492–501. IEEE / ACM, 2013.

[25] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] 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.

[27] A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[28] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, Ø. Haugen, B. Møller-Pedersen, and G. K. Olsen. Developing a software product line for train control: A case study of cvl. In J. Bosch and J. Lee, editors, *SPLC*, volume 6287 of *LNCS*, pages 106–120. Springer, 2010.

[29] E. Tempero, J. Noble, and H. Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *Proc. of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07*, pages 95–104, New York, NY, USA, 2007. ACM.

[31] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC'07*, pages 233–242. IEEE, 2007.

[32] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN Notices*, volume 47, pages 441–454. ACM, 2012.