Refactoring Java Software Product Lines

Jongwook Kim
Iona College
jkim@iona.edu

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

ABSTRACT

Refactoring is a staple of *Object-Oriented* (OO) program development. It should be a staple of OO *Software Product Line* (SPL) development too. X15 is the first tool to support the refactoring of Java SPL codebases. X15 (1) uses Java custom annotations to encode variability in feature-based Java SPLs, (2) projects a view of an SPL product (a program that corresponds to a legal SPL configuration), and (3) allows programmers to edit and refactor the product, propagating changes back to the SPL codebase. Case studies apply 2316 refactorings in 8 public Java SPLs and show that X15 is as efficient, expressive, and scalable as a state-of-the-art feature-unaware Java refactoring engine.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**;

KEYWORDS

refactoring, software product lines

ACM Reference format:
Jongwook Kim, Don Batory, and Danny Dig. 2017. Refactoring Java Software Product Lines. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 10 pages.
https://doi.org/10.1145/3106195.3106201

1 INTRODUCTION

Refactoring has been a staple in *Object-Oriented* (OO) programming for at least a quarter century [24, 45], and a standard tool in *Integrated Development Environments* (IDEs) for at least a decade [19]. *Software Product Lines* (SPLs) have an equally long and rich history [2, 29]. Despite progress, there are *no* tools – research prototypes [3, 6, 32, 37] or commercial tools [14, 38, 48] – that can refactor OO SPLs. In this paper, we present X15, the first tool to refactor Java SPLs.

An SPL is a family of related programs [2]. Amortizing the cost to design and maintain their commonalities makes SPLs economical [2]. Programs of an SPL are distinguished by *features* — increments in program functionality. Each program, henceforth *product*, in an SPL is defined by a unique set of features called a *configuration* [2].

SPLC '17, September 25-29, 2017, Sevilla, Spain

ACM ISBN 978-1-4503-5221-5/17/09…$15.00
https://doi.org/10.1145/3106195.3106201

Variability in an SPL codebase relies on *presence conditions*, a predicate expressed in terms of features that indicates when a fragment of code, declaration, file or package is to be included in an SPL product. A typical use-case is with `#if-#endif` preprocessor constructs: if the presence condition of `#if` is `true` for a configuration, the content that is enclosed by `#if-#endif` is included in the product; otherwise it is erased [2]. The Linux Kernel is a huge SPL, consisting of 8M LOC and over 10K features [42, 54]. It uses the *C-preprocessor* (CPP) to remove code to produce the C codebase for a configuration.

The presence or absence of a feature in Java can be encoded by a global `static boolean` declaration; the Java compiler can evaluate feature predicates to remove unreachable code in `if(feature_exp)` statements. But removing entire declarations such as packages, types, fields, and methods is not possible with existing Java constructs. So Java SPLs are hacked in some manner to achieve this additional and essential effect.

Preprocessing is the standard solution [28, 47, 55], although officially Java shuns preprocessors [18]. Another way is to copy and assemble code fragments from an SPL codebase $\mathbb{P}$ to produce an SPL product $\mathtt{P_C}$ where $\mathtt{C}$ is $\mathtt{P_C}$'s configuration [3, 6, 32, 38]. Both create a separate codebase for $\mathtt{P_C}$ that a programmer edits to improve, tune, and repair $\mathtt{P_C}$. Doing so exposes two critical problems in SPL tooling.

First, given an edited product $\mathtt{P_C}$, how are its edits propagated back to $\mathbb{P}$, the SPL codebase? Early SPL tools [6, 38] had back-propagation capabilities. Now there are more sophisticated tools to back-propagate edits (see [57, 60] for surveys). But none correctly propagates changes from $\mathtt{P_C}$ to $\mathbb{P}$ made by refactorings. Why? Here is a simple example: renaming a field in $\mathtt{P_C}$ is easy, but not all references of the field reside in $\mathtt{P_C}$; other references exist in $\mathbb{P}$ that are *not* in $\mathtt{P_C}$. Thus, back-propagating edits will rename some, but not all, references to a field, breaking $\mathbb{P}$. *In short, refactorings are incompatible with SPL back-propagation tools.*

Second, conditional compilation removes all vestiges of variability from a CPP-infused SPL codebase. In contrast, to refactor a codebase with variability requires the exact knowledge that conditional compilation erases. *Variability-aware compilers* (VACs) – compilers that integrate CPP constructs into the grammar of a host language – are the current solutions to this impasse [11, 16, 33, 61]. VACs generate AST nodes with presence conditions and variability-aware control-flow graphs which are needed for both precondition checks and code transformations of SPL refactorings [40]. But writing a VAC – even for the C-language – is daunting [10, 23, 33]. We are aware of only one VAC for a mainstream OO language, C++ [11]. And even if a VAC exists, we would still need to create a companion refactoring engine for this compiler – yet another daunting task.

We present X15, the first feature-aware refactoring engine for Java that solves the above problems. X15 (i) uses a standard Java compiler, (ii) relies on Java custom annotations to encode SPL variability in a simple and intuitive way, (iii) incorporates code folds of an SPL codebase to produce a 'view' of an SPL product that programmers can edit and refactor, and (iv) behind the curtains X15 applies corresponding edits and feature-aware refactorings to the SPL codebase.

The novel contributions of this paper are:

- The X15 tool for editing, projecting, and refactoring Java SPLs;
- Identifying preconditions that must become feature-aware; and
- Case studies that apply 2316 refactorings in 8 Java SPLs and show X15 is as efficient, expressive, and scalable as a state-of-the-art feature-unaware refactoring engine R3 [36].

2 THE R3 REFACTORING ENGINE

R3 is a new refactoring engine for Java [36]. It is an improvement over the *Eclipse Java Development Tools* (JDT) refactoring engine as (a) it allows programmers to write *refactoring scripts* — programmatic sequences of refactoring invocations — and (b) it executes these scripts 10× faster. R3 uses a form of reflection similar to Java reflection.

Java reflection provides an OO façade to inspect Java bytecode files. Classes, fields, and methods defined in Java bytecode are presented as Class, Field, and Method objects in Java. Semantic information on these objects, such as access modifiers of a method declaration, can be harvested through method calls.

R3 does something analogous: it provides an OO façade to inspect *parse trees* of Java programs. Classes, fields, and methods defined in parse trees are presented as RClass, RField, and RMethod objects in Java. Information on these objects can be harvested via R3 methods.

Unlike Java reflection, R3 allows objects to be created and updated, permitting direct manipulation and restructuring of Java programs. Example: RClass methods are methods to refactor Java classes (e.g., to rename or move) or methods to find related objects (e.g., RField and RMethod objects belonging to that RClass). This enables many Gang-of-Four Design Patterns [20] to be partially or fully automated, written as simple Java methods [35, 36].

Consider the adapter pattern. An *Adapter* is a class that implements all methods of a *Target* interface by invoking methods of an *Adaptee* class. The R3 makeAdapter method of Fig. 1 works by (1) retrieving the package of Target, (2) creating a class named adapterClassName as a concrete class of Target in this package, (3) creating a field named 'adapteeFieldName' of type Adaptee, (4) creating a constructor with an argument that initializes this field, (5) for all methods of Target, create a method stub (which has programmer /*TO DOs*/ for that method), and (6) returning the created Adapter to the makeAdapter caller. Because there are /*TO DOs*/, the adapter pattern is partially automatable [35, 36].

```
// A Member of RInterface class; Target interface is 'this'
RClass makeAdapter(RClass adaptee, String adapterClassName) {
  RPackage pkg = getPackage();
  RClass adapter = pkg.newClass(adapterClassName);
  RField f = adapter.addField(adaptee, "adapteeFieldName");
  adapter.addConstructor(f);
  for (RMethod m : getAllMethods())
      adapter.addMethodStub(m);
  adapter.setInterface(this);
  return adapter;
}
```

Figure 1: An R3 makeAdapter Method.

There are many variations of the Adapter pattern; Fig. 1 is one of several offered by R3. R3 implements 18 of the 24 Gang-of-Four design patterns and 34 distinct pattern-directed refactorings [36].

R3, like other Java refactoring engines, is feature-unaware. As X15 is built on top of R3, it inherits the speed of R3 and the ability of its users to write refactoring scripts to retrofit design patterns into Java codebases.

3 X15 ENCODING OF JAVA SPLS

Every feature-based SPL has a *feature model* (FM) that defines the features of an SPL and their relationships. It is well-known that FMs can be mapped to a propositional formula where features are the boolean variables [2, 5]. Each solution to this formula — a true or false assignment to every variable — defines a combination of features that uniquely identify a product in an SPL. A common name for a solution is a *configuration*.

X15 uses the Java custom annotation type Feature to encode a configuration file. Every feature F of an SPL has a static boolean variable F declared inside Feature whose value indicates whether F is selected (true) or not (false). Fig. 2 shows a Feature declaration with three features X, Y, and Z where X and Y are selected and Z is not. The specified configuration is {X,Y}. Feature.java is generated by a feature model configuration tool [2, 5].

```
@interface Feature {
  static final boolean X = true;
  static final boolean Y = true;
  static final boolean Z = false;
  boolean value();
}
```

Figure 2: The Feature Annotation Type

X15 uses Java's built-in annotations to encode variability that complies with the standard Java grammar. Let $\mathbb{P}$ denote the code base of a Java SPL. Every Java declaration (class, method, field, constructor, initializer) in $\mathbb{P}$ has an optional Feature annotation with a boolean expression of Feature variables.[1] If the expression is true for a configuration, the declaration is present in that configuration's product; otherwise it is not. If a declaration has no Feature annotation, it is included in every product of the SPL.

Fig. 3a shows three declarations: Graphics, Square, and Picture. Interface Graphics belongs to every program of the SPL as it has no Feature annotation. Square is added by feature X. Picture is added whenever a pair of features, Y and Z, are both present.

```
interface Graphics {}

@Feature(X)
class Square implements Graphics {}

@Feature(Y && Z)
class Picture implements Graphics {}
```

(a)

```
@Feature(X)
int i, j, k;
```

(b)

```
@Feature(X)
int i, j;

@Feature(Y)
int k;
```

(c)

Figure 3: Feature Annotations

Fig. 3b shows a declaration of three integer fields i, j, k, all belonging to feature X; the Feature annotation is for the entire line. If fields i and j belong to feature X, and k to feature Y, Fig. 3c is used.

Variability in executable code is written using if(feature_exp) statements. For example, it is common to have different bodies

[1] Package-level annotations in Java are placed in a package-info.java file.

for a single method in an SPL. Suppose features X and Y are never both selected. Fig. 4a is a CPP encoding that introduces at most one declaration of method m in any program; Fig. 4b shows the cascading if-else statements used in X15 to encode the same variability inside one declaration of m.

```
#if(X)    int m() {return 1;}
#elif(Y)  int m() {return 2;}
#else     int m() {return 0;}
#endif
```

(a)

```
int m() {
  if(X)       return 1;
  else if(Y)  return 2;
  else        return 0;
}
```

(b)

Figure 4: Encoding Different Method Bodies

Here's the trick on how X15 works: X15's Feature annotations have different semantics than vanilla Java annotations because X15 *can remove Java declarations.* X15 *parses* $\mathbb{P}$ and looks for the parse tree of Feature.java, from which X15 extracts the boolean value for each feature. These values define the *current configuration* C.

Let P_C be the source of the SPL product with configuration C. Fig. 5 sketches the parse tree of a Feature-annotated class declaration of $\mathbb{P}$. X15 sees the Feature annotation and evaluates the feature expression knowing the current configuration. If the expression is true then X15 pretty-prints the parse tree including Feature annotations (minus code fragments that are configuration-disqualified). If the expression is false, X15 does not pretty-print the parse tree, effectively erasing the declaration. This is how X15 *projects* $\mathbb{P}$ w.r.t. C to produce P_C. X15 *never* changes a parse tree during a projection.

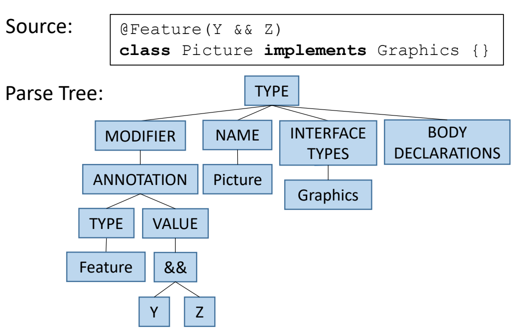

Figure 5: A Parse Tree with an Feature Annotation.

X15 uses two different projections. One projection is sent to the Java compiler to produce bytecodes. The second projection relies on the standard IDE functionality of 'code folding', where code that is not part of P_C is hidden in a code fold. A code fold indicates the location of a *variation point* (VP) — where code in some SPL product is known to exist, but is not present in P_C [2]. Code folds also provide a practical way for programmers to edit code that is visible (i.e., code that belongs to P_C). Programmers can inspect, but *not* edit, folded code. Fig. 6a shows $\mathbb{P}$, Fig. 6b shows P_C when BLUE=false folding BLUE's code, and Fig. 6c shows P_C with unfolded code when BLUE=false.[2]

[2]Of course, there are situations where to correctly edit P_C, programmers must edit $\mathbb{P}$. Suppose a programmer wants to provide a new body to an existing method. To do so, s/he must edit $\mathbb{P}$ to achieve the desired projection. X15 offers a GUI button for users to toggle between editing $\mathbb{P}$ and P_C, should the need arise. [57] has other examples.

```
class A {
 int i=0;
 @Feature(RED)
 int m() {
  if(BLUE) {
   i=i*2;
  }
  return i++;
 }
}
```

(a) SPL Codebase

```
class A {
 int i=0;
 @Feature(RED)
 int m() {
  //[VP]
  return i++;
 }
}
```

(b) BLUE = false (folded)

```
class A {
 int i=0;
 @Feature(RED)
 int m() {
  //[VP]
   if(BLUE) {
    i=i*2;
   }
  //[VP_Ends]
  return i++;
 }
}
```

(c) BLUE = false (expanded)

Figure 6: Code Folding in X15.

Together, both projections provide a useful end-user functionality: an X15 user can see and edit a 'view' (projection) of P_C, the SPL product of the current configuration. Further, s/he can compile P_C and debug it through the code-folded projection, giving the impression that the X15 user is editing, debugging, and developing a single product P_C, even though behind the curtains edits are being made directly to $\mathbb{P}$.

3.1 Refactorings Are Not Edits

If refactorings were just text edits, we would be done. A programmer invokes a refactoring on product P_C, the code of P_C is changed and the edits are copied to $\mathbb{P}$. End of story.

The problem is that *refactorings are more than text edits.* Consider the SPL codebase $\mathbb{P}$ of Fig. 7a. The separate codebase P_X for configuration {X} is Fig. 7b. Fig. 7c shows P_X after renaming Grafix to Graphics. The problem is evident in Fig. 7d: propagating *text* changes made to P_X back to $\mathbb{P}$ breaks $\mathbb{P}$ because not all occurrences of Grafix in $\mathbb{P}$ are renamed to Graphics — the program for configuration {Y} no longer compiles.

```
abstract class Grafix {...}

@Feature(X)
class Square
  implements Grafix {...}

@Feature(Y)
class Picture
  implements Grafix {...}
```

(a) Codebase $\mathbb{P}$

```
abstract class Grafix {...}

@Feature(X)
class Square
  implements Grafix {...}
```

(b) Codebase P_X

```
abstract class Graphics
{...}
@Feature(X)
class Square
  implements Graphics {...}
```

(c) Rename-refactored P_X

```
abstract class Graphics
{...}
@Feature(X)
class Square
  implements Graphics {...}

@Feature(Y)
class Picture
  implements Grafix {...}
```

(d) Code-backpropagation to $\mathbb{P}$

Figure 7: Problems in Refactoring Separate Codebases.

In a nutshell, *back-propagation of text edits is incompatible with refactorings; refactorings make changes to $\mathbb{P}$ that are not part of the codebase of a single product* P_X. Dig and Johnson demonstrated an analogous problem for version control [13]. A new approach is needed to solve this problem, which is discussed next.

4 ALGEBRAS OF FEATURE COMPOSITIONS

Features have long been recognized as the conceptual building blocks of SPL products. Early research (AHEAD [6], FeatureHouse [3], DOP [51]) developed algebras for feature compositions to define the abstract properties of features. Tools were then built to implement these algebras to demonstrate a scientific way to compositionally construct SPL programs. Further, these tools invented

OO language extensions to define concrete feature modules. The ideas behind these language extensions — role-based programming, mixin-layers, and context-oriented programming — have been widely explored. Other research demonstrated how annotated SPL codebases could be mapped to feature modules, and vice versa [2, 30, 31], effectively demonstrating the Turing-equivalence of feature modules and feature annotation implementations of SPLs. *In short, any theorem that can be proven using feature algebras should hold for all feature-based SPL implementations.*

In the following sections, we sketch known ideas and then present the insight that made X15 possible.

4.1 Sum and Projection of Feature Modules

A *feature module* $\mathtt{F_i}$ encapsulates the implementation of feature $\mathtt{i}$. Product $\mathtt{P_C}$ with configuration $\mathtt{C}$ is produced by summing the modules of its features [3, 6, 51]. Thus, if $\mathtt{C} = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$ where X, Y, and Z are features, product $\mathtt{P_C}$ is:

$$\mathtt{P_C} = \sum_{\mathtt{i}\in\mathtt{C}} \mathtt{F_i} = \mathtt{F_X} + \mathtt{F_Y} + \mathtt{F_Z} \tag{1}$$

Let $\mathbb{F}$ be the set of all features. The codebase $\mathbb{P}$ of an SPL is:[3]

$$\mathbb{P} = \sum_{\mathtt{i}\in\mathbb{F}} \mathtt{F_i} \tag{2}$$

Projection, as discussed in Section 3, is a complementary operation to summation because it eliminates feature modules. The C-projection of $\mathbb{P}$ yields $\mathtt{P_C}$:

$$\Pi_{\mathtt{C}}(\mathbb{P}) = \mathtt{P_C} \tag{3}$$

Let $\mathtt{C_1}$ and $\mathtt{C_2}$ be different sets of features from the same SPL (i.e., $\mathtt{C_1}, \mathtt{C_2} \subseteq \mathbb{F}$). An axiom that relates projection and summation is:

$$\Pi_{\mathtt{C_1}}\Big(\sum_{\mathtt{i}\in\mathtt{C_2}} \mathtt{F_i}\Big) = \sum_{\mathtt{i}\in\mathtt{C_1}\cap\mathtt{C_2}} \mathtt{F_i} \tag{4}$$

Equation (1) follows from (2), (3), and (4):

$$\begin{aligned}
\mathtt{P_C} &= \Pi_{\mathtt{C}}(\mathbb{P}) && // (3)\\
&= \Pi_{\mathtt{C}}\Big(\sum_{\mathtt{i}\in\mathbb{F}} \mathtt{F_i}\Big) && // (2)\\
&= \sum_{\mathtt{i}\in\mathtt{C}\cap\mathbb{F}} \mathtt{F_i} && // (4)\\
&= \sum_{\mathtt{i}\in\mathtt{C}} \mathtt{F_i} && // \text{ where } \mathtt{C} \subseteq \mathbb{F}
\end{aligned}$$

As said in Section 3, X15 implements projection in two different ways: $\Pi^{\mathtt{fold}}_{\mathtt{C}}$ code-folds $\mathbb{P}$ to expose only the code of $\mathtt{P_C}$ for viewing, editing and refactoring. $\Pi^{\mathtt{comment}}_{\mathtt{C}}$ comments-out unnecessary code which is then fed to the Java compiler to produce bytecodes for $\mathtt{P_C}$; this compiled version enables programmers to execute, debug, and step-through the code folded version of $\mathtt{P_C}$.

4.2 Theorem for Refactoring SPLs

Let $\mathcal{R}$ be a refactoring. If we $\mathcal{R}$-refactor $\mathtt{P_C}$, we get $\mathtt{P^{\mathcal{R}}_C}$:

$$\mathcal{R}(\,\Pi_{\mathtt{C}}(\mathbb{P})\,) = \mathcal{R}(\mathtt{P_C}) = \mathtt{P^{\mathcal{R}}_C} \tag{5}$$

[3] A common name for $\mathbb{P}$ is a *150% design* – it includes all possibilities.

As $\mathcal{R}$ changes $\mathtt{P_C}$, $\mathcal{R}$ must also change $\mathbb{P}$. But how? Our conjecture and theorem is this: $\mathtt{P^{\mathcal{R}}_C}$ can be computed by the $\mathcal{R}$-refactoring of $\mathbb{P}$ followed by a C-projection:

$$\Pi_{\mathtt{C}}(\,\mathcal{R}(\mathbb{P})\,) = \mathtt{P^{\mathcal{R}}_C} \tag{6}$$

Equivalently, (6) is the commuting diagram of Fig. 8 where the operations of projection and refactoring commute [46].

$$\begin{tikzcd}
\mathbb{P} \arrow[r, "\Pi_C"] \arrow[d, "\mathcal{R}"'] & P_C \arrow[d, "\mathcal{R}"] \\
\mathbb{P}^{\mathcal{R}} \arrow[r, "\Pi_C"] & P_C^{\mathcal{R}}
\end{tikzcd}$$

Figure 8: Theorem of SPL Refactoring.

SPL programmers must realize that refactoring an SPL codebase $\mathbb{P}$ has more constraints than just refactoring a single product $\mathtt{P_C}$. We explain in Section 5.2 that the preconditions to $\mathcal{R}$-refactor $\mathbb{P}$ imply the preconditions to $\mathcal{R}$-refactor $\mathtt{P_C}$. Our proof of (6) assumes the preconditions to $\mathcal{R}$-refactor $\mathbb{P}$ are satisfied. Therefore in (6), $\mathcal{R}$ represents the code transformation that is made by an $\mathcal{R}$-refactoring; $\mathcal{R}$ cannot be applied if any of its preconditions fail.

Here is our key insight about refactoring SPLs: common refactorings are largely oblivious to feature module boundaries. That is, when a program $\mathtt{P} = \mathtt{A} + \mathtt{B}$ is $\mathcal{R}$-refactored, one expects both modules A and B to be modified by $\mathcal{R}$, namely $\mathtt{P^{\mathcal{R}}} = \mathtt{A^{\mathcal{R}}} + \mathtt{B^{\mathcal{R}}}$.

> **Example:** Method m in Fig. 9 is defined in class/feature A. Class/Feature B calls m. When m is renamed to n, both features A and B are modified to $\mathtt{A^{\mathcal{R}}}$ and $\mathtt{B^{\mathcal{R}}}$.

```
@Feature(A)
class A {
  void m() {}
}

@Feature(B)
class B {
  void foo(A a) { a.m(); }
}
```

(a) Before

```
@Feature(A)
class A {
  void n() {}
}

@Feature(B)
class B {
  void foo(A a) { a.n(); }
}
```

(b) After Renaming m to n

Figure 9: Rename-Method Refactoring

This insight translates into a new axiom for feature algebras: The $\mathcal{R}$-refactoring of a sum of feature modules A and B equals the sum of the $\mathcal{R}$-refactored modules:

$$\mathcal{R}(\mathtt{A} + \mathtt{B}) = \mathcal{R}(\mathtt{A}) + \mathcal{R}(\mathtt{B}) \tag{7}$$

The proof of (6) follows from (5) and (7):

$$\begin{aligned}
\Pi_{\mathtt{C}}(\,\mathcal{R}(\mathbb{P})\,) &= \Pi_{\mathtt{C}}\Big(\,\mathcal{R}\Big(\sum_{\mathtt{i}\in\mathbb{F}} \mathtt{F_i}\Big)\,\Big) && // \text{ by } (2)\\
&= \Pi_{\mathtt{C}}\Big(\sum_{\mathtt{i}\in\mathbb{F}} \mathcal{R}(\mathtt{F_i})\Big) && // \text{ by } (7)\\
&= \sum_{\mathtt{i}\in\mathtt{C}} \mathcal{R}(\mathtt{F_i}) && // \text{ by } (4)\\
&= \mathcal{R}\Big(\sum_{\mathtt{i}\in\mathtt{C}} \mathtt{F_i}\Big) && // \text{ by } (7)\\
&= \mathcal{R}(\mathtt{P_C}) && // \text{ by } (5)\\
&= \mathtt{P^{\mathcal{R}}_C} && // \text{ by } (5)
\end{aligned}$$

Equation (6) tells us how to translate refactorings of SPL products, namely refactorings to $\mathtt{P_C}$*, to refactorings of the SPL codebase* $\mathbb{P}$*. When* an X15 user applies a refactoring $\mathcal{R}$ to P_C, s/he sees $\mathcal{R}(P_C) = P_C^{\mathcal{R}}$ as the result. But behind the curtains, X15 is really applying $\mathcal{R}$ to $\mathbb{P}$, and taking its C-projection to present $P_C^{\mathcal{R}}$ to the programmer.[4]

> **Example:** An X15 user renames Grafix to Graphics in P_X of Fig. 7. X15 applies this refactoring to the entire codebase $\mathbb{P}$. The result is that *all* references to Grafix are renamed to Graphics and that the resulting projection (view) of P_X is correct as in Fig. 7c. X15 updates *all* programs in an SPL that are affected by this rename, thus keeping $\mathbb{P}$ consistent.

5 REFACTORING PRECONDITIONS

We begin by reviewing a fundamental SPL analysis, and then show how this analysis is relevant to refactoring preconditions.

5.1 Safe Composition

Safe Composition (SC) is a common SPL analysis. It is the verification that every program of an SPL compiles without error [2, 12, 33, 34, 44, 58].

Suppose that field x is added by feature X, field y is added by feature Y, and statement "x = y;" is added by feature F. This relationship is expressed by the presence condition $\psi := (F \Rightarrow X \wedge Y)$. That is, when statement "x = y;" appears in a product, so too must the declarations for x and y.

Let ϕ be the propositional formula of the SPL's FM [2, 5]. If $\phi \wedge \neg\psi$ is satisfiable, then at least one SPL product does not satisfy ψ and hence will not compile [12]. Similarly, *dead code* is source that appears in no SPL product. Let δ be the presence condition for code fragment ℓ. If $\phi \wedge \delta$ is unsatisfiable, then ℓ is dead code.

X15 culls $\mathbb{P}$ for all distinct ψ and δ and verifies that no program in the SPL violates either constraint. We say $\mathbb{P}$ *satisfies* SC if no presence condition ψ is violated and $\mathbb{P}$ is *dead code free* if no dead code fragments are found. X15 uses GUIDSL of the AHEAD tool suite [6], which in turn uses SAT4J [49], to solve SAT problems.

5.2 Preconditions for SPL Refactorings

Theorem (6) assumes the preconditions for $\mathcal{R}$-refactoring $\mathbb{P}$ are satisfied. But what are these preconditions? Consider this example: A programmer wants to refactor the base product P_{base} whose SPL codebase $\mathbb{P}$ is Fig. 10. Method bar is invisible to the programmer as it belongs to unselected feature X. If the programmer tries to rename foo to bar, the rename fails since there is at least one product in the SPL (any configuration with X) where this rename fails, even though renaming foo to bar in P_{base} is legal. We use the rule of Liebig, et al. [40]: *An $\mathcal{R}$-refactoring of an SPL fails if $\mathcal{R}$ fails on any product of that SPL.*

```
class A {
  void foo() {}

  @Feature(X)
  void bar() {}
}
```

Figure 10: Renaming foo to bar Fails

X15 reports precondition failures of a refactoring $\mathcal{R}$ by citing a condition or SPL configuration where it fails. This is done by '*lifting*' a refactoring precondition to a SC constraint ψ and verifying all SPL products satisfy ψ. (By definition the lifted constraint implies the precondition on program P_C.) R3 supports 34 different primitive refactorings and uses 39 distinct primitive precondition checks, where each R3 refactoring uses a subset of these 39 checks. X15 supports all of R3's primitive refactorings and preconditions.

We expected most R3 preconditions would be feature-aware, but were surprised when only 5 of the 39 required lifting. Why?

(1) Java annotations cannot be attached to *any* code fragment, such as a Java modifier. Thus, preconditions dealing with modifiers are not lifted, and thus remain identical to their unlifted R3 counterparts. And
(2) Some preconditions are feature-independent, such as Declaring Type[5] and Constructor,[6] so lifting them is unnecessary.

Here are the preconditions that required lifting:

- **Binding Resolution.** Before a method is moved, a lifted check is performed: Let $\rho(e)$ be the presence condition for program element e. Then, $\rho(A.m)$ is the presence condition for method A.m, and $\rho(B.m)$ is the presence condition for m after it is moved to class B. For every reference r to A.m before the move we know $\rho(r) \Rightarrow \rho(A.m)$. The condition to verify after the move is for every reference r: $\rho(r) \Rightarrow \rho(B.m)$. In Fig. 11, $\rho(r) = \mathrm{YELLOW}$, $\rho(A.m) = \mathrm{BLUE} \wedge \mathrm{GREEN}$ and $\rho(B.m) = \mathrm{RED} \wedge \mathrm{GREEN}$. A similar lifted check verifies that all declarations referenced in A.m are still present for B.m to reference.

```
@Feature(BLUE)
class A {
  @Feature(GREEN)
  void m(B b) {}
}

@Feature(RED)
class B {
}

@Feature(YELLOW)
class C {
  void n(A a, B b) {
    a.m(b)   // r
  }
}
```

(a) Before

```
@Feature(BLUE)
class A {
}

@Feature(RED)
class B {
  @Feature(GREEN)
  void m(A a) {}
}

@Feature(YELLOW)
class C {
  void n(A a, B b) {
    b.m(a)   // r
  }
}
```

(b) After Moving A.m to B.m

Figure 11: Binding Resolution Constraint

- **Execution Flow.** Fig. 12a shows a feature-unaware class A. It is illegal to inline method n due to the return statement inside n, as the i++ statement of method m would never be executed after refactoring. In contrast, inlining is allowed in class A of Fig. 12b, provided that feature BLUE implies ¬ RED. Although this example seems artificial, we did encounter it in Section 6.

```
class A {
  int i = 0;

  void m() {
    n();
    i++;
  }

   void n() {
    i = 1;
    return;
  }
}
```

(a) Without Features

```
class A {
  int i = 0;

   void m() {
    if(BLUE) n();
    if(RED) i++;
  }

   void n() {
    i = 1;
    return;
  }
}
```

(b) With Features

Figure 12: Inlining Constraint

[4]Equation (6) also tells us that algebras for feature summation and refactoring are elegant. The name of this algebraic structure is a 'left M-semimodule over a monoid' [26].

[5]A method cannot be moved if its enclosing type is an annotation or interface.
[6]A constructor cannot be moved.

- **Variable Capture.** Renaming field `B.j` to `B.i` in Fig. 13a intercepts the binding to inherited variable `A.i` and an error is reported. In Fig. 13b, capture does not arise if features `BLUE` and `RED` are mutually exclusive [40].

```
class A {

  int i = 0;
}

class B extends A {

  int j = 1;


  void m() { i++; }
}
```

(a) Without Features

```
class A {
  @Feature(BLUE)
  int i = 0;
}

class B extends A {
  @Feature(RED)
  int j = 1

  @Feature(BLUE)
  void m() { i++; }
}
```

(b) With Features

Figure 13: Variable Capturing Constraint

- **Explicit Super Invocation.** Default constructors are needed in class inheritance hierarchies. Consider Fig. 14a. If feature `BLUE` is unselected, Java generates an error because class `A` has no default constructor.
- **Single Constructor Call.** A singleton design pattern refactoring introduces a single static instance of a class `A`, and replaces the only constructor call to `A` in a program with a reference to this instance. The program in Fig. 14b satisfies the singleton constraint provided that features `BLUE` and `RED` are mutually exclusive.

```
class A {
  A(int i) {}

  @Feature(BLUE)
  A() {}
}

class B extends A {}
```

(a) Non-Default Constructor Constraint

```
class A {
  A(){ /*do something*/ }

  @Feature(BLUE)
  A a = new A();

  @Feature(RED)
  String s = new A().toString();
}
```

(b) Singleton Constraint

Figure 14: Other Constraints

5.3 Implementation Notes

Feature models of SPLs are rather static; they do change but slowly. `X15` culls $\mathbb{P}$ for constraints which are translated to a large number of SAT problems to solve. From experimental results in Section 6, a crude estimate is about 1 SAT check per every 2 lines of source. A saving grace is that the number of unique SAT checks is small, possibly orders of magnitude smaller than the crude estimate [58].

`X15` leverages the stability of an SPL's feature model by caching the results of SAT checks. That is, a set of (SC ψ or δ, boolean-validity) pairs is stored. When a feature-aware condition arises, `X15` identifies the unique SAT checks to verify, and looks in its SAT cache. Only when a previously unseen SAT check is encountered will a SAT solver be invoked, and of course, its result is henceforth cached. The cache is cleared whenever the feature model is updated.

6 EVALUATION

We evaluated `R3` by demonstrating that its scripts could retrofit design patterns into real-world programs [36]. The focus of the evaluation was on patterns/scripts that (a) were the hardest to manually create and that (b) executed the most refactorings and the greatest number of different types of refactorings. These were the `makeVisitor` and `inverseVisitor` scripts.

To motivate `inverseVisitor`, one can imagine creating a Visitor to inspect a family of related methods as part of some debugging process, where some `visit` methods are updated. Eventually, the Visitor is removed and the updated methods are returned to their original classes. `inverseVisitor` is not a rollback (which would remove all method updates); rather, it is a refactoring script that preserves method updates in a Visitor removal [35, 36].

We use these same `makeVisitor` and `inverseVisitor` scripts to compare `X15`'s performance w.r.t.`R3`. `X15` has the same expressivity as `R3` — except of course in an SPL context. Like `R3`, `X15` supports 18 of the 23 design patterns in the Gang-of-Four text [20]; the other 5 patterns do not benefit from automation [36].

We answer three research questions:

- **RQ1:** Can `X15` refactor Java SPLs?
- **RQ2:** How fast is `X15` compared with `R3`?
- **RQ3:** How is performance improved by caching SAT checks?

6.1 Experimental Set-Up

We selected 8 public Java SPLs for our studies that are widely-used for product-line analyses [56]. Column **Applications** of Table 1 lists the eight target SPLs along with their lines of code, number of regression tests, and number of features. Three SPLs (**A**HEAD, **C**alculator, and **E**levator) had regression tests that could validate `X15` refactorings. Two (**N**otepad and **S**udoku) lacked regression tests but could be checked by manually invoking their GUIs before and after running `X15` scripts to verify behavior preservation. The remaining three (**L**ampiro, **M**obileMedia, and **P**revayler) also lacked regression tests. We did not know how to execute these programs, so we could only verify that they compiled without errors before and after refactoring.

Table 1: Applications

Applications	ID	LOC	# of Tests	# of Features
AHEAD Mixin	**A**	26K	56	16
Calculator	**C**	312	17	6
Elevator	**E**	973	6	6
Notepad	**N**	1192	21*	27
Sudoku	**S**	1975	22*	6
Lampiro	**L**	44K	0	16
Mobile Media	**M**	4653	0	7
Prevayler	**P**	8009	0	5

– * indicates the regression tests done by invoking user interface operations manually.

We used an Intel CPU `i7-2600 3.40`GHz, `16` GB main memory, Windows `7 64`-bit OS, and Eclipse JDT `4.4.2` (Luna) in our work. Execution times were measured by `VisualVM` (ver. `1.3.8`) [59]. Each experiment was executed five times and the average is reported.

6.2 Results

6.2.1 *Table Organization.*

Table 2 shows the results of `makeVisitor`, a refactoring that introduces a visitor pattern by moving identical-signature methods in a class hierarchy into a newly-created visitor class (see [20, 36] for details). Each row is an experiment that creates a Visitor for a particular method; different rows use different methods. These methods are simply identified by a

Table 2: `makeVisitor` Results

M#	# of Refs	R3 Time (sec)	X15 Time (sec)					Overhead	
			Pred Coll	Use SAT Caching?					
				No	Yes	No	Yes	No	Yes
		(R3T)	(α)	Ext Prec (γ)		Tot (X15T)		X15T-R3T	
A1	54	2.01		0.09 [104]	0.03 [2]	3.07	3.01	1.05	1.00
A2	56	1.91		0.07 [56]	0.03 [2]	2.96	2.91	1.04	0.99
A3	58	2.28	0.96	0.11 [133]	0.04 [9]	3.36	3.29	1.07	1.00
A4	124	2.07		0.08 [128]	0.03 [3]	3.13	3.07	1.05	0.99
A5	552	3.64		0.17 [287]	0.04 [3]	4.77	4.64	1.13	1.00
C1	4	0.23		0.03 [17]	0.03 [3]	0.31	0.31	0.08	0.07
C2	4	0.25	0.04	0.03 [4]	0.03 [4]	0.33	0.33	0.07	0.07
C3	4	0.24		0.02 [3]	0.03 [3]	0.31	0.32	0.07	0.07
E1	4	0.36		0.02 [3]	0.03 [2]	0.46	0.47	0.10	0.11
E2	4	0.35	0.08	0.02 [4]	0.03 [2]	0.46	0.47	0.11	0.11
E3	4	0.36		0.02 [3]	0.02 [2]	0.47	0.46	0.11	0.10
N1	4	0.59		0.06 [28]	0.03 [3]	0.78	0.75	0.19	0.16
N2	4	0.69	0.13	0.03 [2]	0.03 [2]	0.85	0.85	0.16	0.16
N3	4	0.67		0.03 [5]	0.03 [3]	0.83	0.84	0.16	0.16
S1	4	0.38		0.03 [8]	0.03 [1]	0.57	0.56	0.19	0.18
S2	4	0.38	0.15	0.03 [18]	0.04 [1]	0.57	0.57	0.19	0.19
S3	6	0.39		0.04 [47]	0.03 [2]	0.59	0.58	0.20	0.18
L1	16	3.28		0.09 [147]	0.03 [1]	4.04	3.97	0.75	0.69
L2	26	2.86		1.16 [937]	0.05 [3]	4.69	3.57	1.82	0.71
L3	26	3.33	0.66	0.10 [207]	0.03 [1]	4.10	4.02	0.76	0.69
L4	32	2.89		0.87 [723]	0.05 [1]	4.43	3.60	1.53	0.71
L5	42	3.62		1.16 [1294]	0.06 [2]	5.44	4.34	1.82	0.72
M1	6	0.65		0.07 [79]	0.03 [2]	0.87	0.82	0.21	0.17
M2	6	0.63		0.04 [16]	0.03 [3]	0.81	0.80	0.18	0.17
M3	6	0.63	0.14	0.04 [14]	0.03 [3]	0.81	0.80	0.18	0.17
M4	8	0.65		0.07 [76]	0.02 [2]	0.86	0.81	0.21	0.16
M5	34	0.75		0.19 [432]	0.06 [18]	1.08	0.95	0.33	0.20
P1	10	0.92		0.04 [26]	0.03 [1]	1.28	1.27	0.36	0.35
P2	10	0.92		0.03 [26]	0.03 [1]	1.28	1.27	0.36	0.35
P3	10	0.95	0.32	0.03 [22]	0.04 [3]	1.31	1.31	0.36	0.36
P4	16	0.95		0.03 [17]	0.02 [1]	1.31	1.30	0.35	0.34
P5	16	0.95		0.03 [17]	0.03 [1]	1.31	1.30	0.36	0.35

– **M#** is an identifier of a method from Application ID **M** in Table 1.
– N of [N] is the # of SAT problems solved for extra precondition checks.

Table 3: `inverseVisitor` Results

M#	# of Refs	R3 Time (sec)	X15 Time (sec)					Overhead	
			Pred Coll	Use SAT Caching?					
				No	Yes	No	Yes	No	Yes
		(R3T)	(α)	Ext Prec (γ)		Tot (X15T)		X15T-R3T	
A1$_V$	54	1.99	0.91	0.14 [208]	0.02 [2]	3.04	2.93	1.05	0.93
A2$_V$	56	1.92	0.92	0.09 [112]	0.03 [2]	2.94	2.87	1.02	0.95
A3$_V$	58	2.24	0.91	0.12 [191]	0.04 [7]	3.28	3.19	1.03	0.95
A4$_V$	124	1.95	0.87	0.16 [254]	0.03 [3]	2.98	2.85	1.03	0.90
A5$_V$	552	3.08	0.69	0.35 [841]	0.05 [3]	4.13	3.84	1.04	0.75
C1$_V$	4	0.21	0.02	0.04 [19]	0.03 [3]	0.27	0.26	0.06	0.05
C2$_V$	4	0.23	0.02	0.04 [9]	0.03 [4]	0.30	0.28	0.06	0.05
C3$_V$	4	0.22	0.04	0.03 [4]	0.03 [3]	0.30	0.30	0.08	0.07
E1$_V$	4	0.33	0.07	0.03 [6]	0.02 [2]	0.43	0.42	0.10	0.09
E2$_V$	4	0.33	0.07	0.03 [6]	0.03 [2]	0.43	0.43	0.10	0.10
E3$_V$	4	0.33	0.05	0.03 [6]	0.02 [2]	0.42	0.42	0.08	0.08
N1$_V$	4	0.54	0.12	0.06 [28]	0.04 [4]	0.73	0.71	0.18	0.16
N2$_V$	4	0.30	0.12	0.03 [3]	0.03 [3]	0.46	0.46	0.15	0.16
N3$_V$	4	0.33	0.12	0.04 [6]	0.04 [3]	0.49	0.49	0.16	0.16
S1$_V$	4	0.39	0.17	0.03 [9]	0.03 [1]	0.59	0.59	0.20	0.20
S2$_V$	4	0.39	0.17	0.03 [9]	0.03 [1]	0.60	0.60	0.20	0.21
S3$_V$	6	0.43	0.17	0.05 [65]	0.03 [3]	0.65	0.64	0.22	0.20
L1$_V$	16	3.28	0.63	0.08 [90]	0.02 [1]	3.99	3.94	0.71	0.66
L2$_V$	26	2.87	0.63	0.24 [552]	0.05 [3]	3.75	3.56	0.88	0.68
L3$_V$	26	3.32	0.63	0.10 [165]	0.03 [1]	4.06	3.99	0.74	0.67
L4$_V$	32	2.93	0.63	0.30 [780]	0.05 [1]	3.86	3.61	0.93	0.68
L5$_V$	42	3.73	0.66	0.39 [1020]	0.05 [1]	4.79	4.45	1.05	0.71
M1$_V$	6	0.64	0.15	0.07 [92]	0.03 [6]	0.86	0.82	0.22	0.18
M2$_V$	6	0.60	0.16	0.05 [18]	0.03 [6]	0.82	0.80	0.21	0.19
M3$_V$	6	0.62	0.15	0.04 [18]	0.03 [6]	0.81	0.80	0.19	0.18
M4$_V$	8	0.66	0.16	0.09 [93]	0.04 [2]	0.91	0.86	0.25	0.20
M5$_V$	34	0.82	0.15	0.26 [549]	0.06 [20]	1.24	1.03	0.41	0.21
P1$_V$	10	0.80	0.37	0.04 [37]	0.02 [1]	1.21	1.19	0.41	0.39
P2$_V$	10	0.80	0.36	0.04 [37]	0.03 [1]	1.20	1.19	0.40	0.39
P3$_V$	10	0.86	0.39	0.03 [27]	0.03 [3]	1.28	1.28	0.42	0.42
P4$_V$	16	0.89	0.35	0.04 [27]	0.03 [1]	1.28	1.27	0.39	0.38
P5$_V$	16	0.86	0.36	0.04 [27]	0.03 [1]	1.26	1.25	0.40	0.39

– **M#**$_V$ is the application where a visitor is to be removed to reproduce **M#** of Table 2.
– N of [N] is the # of SAT problems solved for extra precondition checks.

number (**M#**).[7] The second column, **# of Refs**, is the total number of refactorings executed to make a Visitor for that experiment.

Each of our SPLs has a 'max' configuration – all features are selected. We let R3 execute the same refactoring script on the 'max' configuration product of each SPL to estimate the overhead of X15 w.r.t. R3. The average execution time for R3 is **R3 Time** (R3T).

The columns below list the computation times for feature-aware refactorings in X15:

- **Pred Coll** (α): time to collect presence conditions on all declarations and references.
- **Ext Prec** (γ): time spent on feature-aware precondition checks (with/without SAT-caching), including the time for caching SAT solutions.
- **Tot** (X15T): the total X15 execution time, (R3T)+(α)+(γ), with-/without SAT-caching.

By comparing the total times using R3 and X15, we estimate the overhead of feature-aware refactorings in our experiments, the subject of the last column:

- **Overhead**: the overhead difference (X15T) – (R3T) in terms of execution time with/without caching.

Table 3 lists the results of `inverseVisitor` in an identical tabular structure. Although the total number of refactorings needed for `inverseVisitor` is equal to that of `makeVisitor`, the set of refactorings invoked and corresponding X15 scripts are different and the number of SAT problems to solve for `inverseVisitor` is slightly larger than that of `makeVisitor`.

6.2.2 *Answers to Research Questions.*

RQ1: Can X15 refactor Java SPLs? X15 successfully retrofitted 64 design pattern instances on our SPLs using a total of 2316 refactorings: 32 experiments added a visitor pattern and 32 removed a visitor. The most challenging experiment, **A**5, executed 552 primitive refactorings. Other experiments required fewer as their visitor class had fewer '`visit`' methods.

Our conclusion: X15 can indeed refactor SPL codebases.

RQ2: How fast is X15 compared with R3? To answer this question, we used three measures:

(1) Consider the execution times for X15 for all `makeVisitor` and `inverseVisitor` experiments. The largest execution time for X15, experiment **A**5, took 4.64 seconds using cached SAT solutions. The comparable experiment using R3 took 3.64 seconds. (For a perspective on R3's improvement over the Eclipse JDT engine, a comparable refactoring to **A**5 took Eclipse 298 seconds to execute, a speedup of over 100× [36].)
Without caching, row **L**5 took 5.44 seconds; the comparable experiment using R3 took 3.62 seconds. The numbers for `inverseVisitor` in Table 3 are similar. For less demanding scripts — remember: *rows are* not *individual refactorings* — all

[7] Each method name of **M#** is getAST_Exp, getBlock, checkForErrors, getQName, printorder, polishCLI, printSupportedOps, getEvalResult, getExecutedActions, checkOnlySpecification, isAbortedRun, neW, print, fCenter, setFieldPrivate, updateSudokuViews, trySolve, setDirty, packetReceived, getHeight, paint, keyPressed, resetRecordStore, getByteFromMediaInfo, getMeiaArrayOfByte, goToPreviousScreen, handleCommand, executeAndQuery*, executeAndQuery*, receive, close, createTestConnection. *These methods are distinct but have identical names.

X15 executions complete in under 1.4 seconds; the corresponding R3 executions finish in under 1 second. On average across all experiments, X15 was 0.5 seconds slower than R3 per experiment.

(2) R3 harvests information from $\mathbb{P}$ before it executes a script. X15 must collect more information; specifically feature presence predicates (see column α of Table 2 and Table 3). This adds one more second of execution time for the largest SPLs. For a perspective, between the time a user clicks the Eclipse GUI and the list of available scripts is displayed, both R3 and X15 harvesting can be done with time to spare.

(3) Over 80% of Eclipse refactoring execution time is consumed by checking preconditions [36]. In contrast, R3 precondition checking is almost instantaneous [36]. X15 takes advantage of R3's speed, but spends extra time for feature-aware precondition checks (see column (γ)). In the largest SPLs, this adds another 1.2 seconds without SAT-caching. For smaller SPLs, the additional time is unnoticeable.

Our conclusion: X15 refactors SPLs at comparable speeds to R3, a state-of-the-art feature-unaware refactoring engine.

RQ3: How is performance improved by caching SAT checks? To answer this question, we used two measures:

(1) The average overhead for checking feature-aware preconditions in the makeVisitor experiment was 0.52 seconds without caching SAT solutions. With caching, the average overhead dropped to 0.40 seconds. For a perspective, experiment **L**5 spent 1.16 seconds proving 1,294 SAT problems, a vast majority of which were duplicates. With caching, only one extra theorem required a SAT proof, taking 0.06 seconds.

(2) Table 4 shows the time and number of SAT problems for dead code and SC checks on the SPLs in Table 1. Again, we took two different approaches (non-caching and caching) to measure how much time X15 can save by reusing SAT solutions. On average for our experiments, caching increased the speed of dead code checks by 1.03× and SC by 15×. SC benefits from caching much more than dead code because SC solves a larger number of SAT problems so it more likely reuses SAT solutions.

Table 4: Dead Code and Safe Composition Check Results

App ID	No-caching (seconds)		Caching (seconds)		Speed Up	
	DC	SC	DC	SC	DC	SC
A	1.18 [182]	94.68 [19,811]	1.13 [176]	4.21 [62]	1.04 (6)	22.46 (19,749)
C	0.11 [42]	0.14 [108]	0.10 [39]	0.08 [9]	1.10 (3)	1.75 (99)
E	0.23 [158]	0.26 [676]	0.23 [155]	0.13 [16]	1.00 (3)	1.97 (660)
N	0.38 [188]	0.50 [635]	0.47 [188]	0.24 [86]	0.81 (0)	2.07 (549)
S	0.29 [79]	0.43 [854]	0.30 [64]	0.25 [14]	0.95 (15)	1.70 (840)
L	0.78 [138]	6.74 [29,501]	0.68 [62]	1.26 [11]	1.15 (76)	5.35 (29,490)
M	0.36 [125]	0.87 [1,976]	0.27 [87]	0.22 [25]	1.34 (38)	3.95 (1,951)
P	0.45 [94]	1.24 [3,329]	0.47 [88]	0.47 [12]	0.95 (6)	2.65 (3,317)

– [N] is the # of SAT checks solved.
– (N) is the # of SAT checks whose solution was found in the cache.
– DC stands for Dead Code checks.

On average, the overhead for feature-awareness in inverseVisitor refactorings was 0.45 seconds without caching and 0.38 seconds with caching, which is miniscule. The results of inverseVisitor are no different than those of makeVisitor.

Readers may be surprised at the low execution time for SAT checks. This is because the feature models of our SPLs are relatively simple; all have a small number of features.

Our conclusion: caching solutions to SAT checks does indeed improve performance and will be even more important for complex feature models.

6.3 Converting SPL Codebases to X15 Format

Every SPL tool today uses a unique means to encode variability.[8] In order to use these SPLs in our experiments, we had to modify them to use X15 annotations.

SPLs that used AHEAD [6] and FeatureHouse [3], namely **M**ixin, **C**alculator and **E**levator, were partially translated by tools – manual work was still needed. The remaining five applications (**N**otepad, **S**udoku, **L**ampiro, **M**obileMedia, and **P**revayler) used CIDE [32], which could be transformed into javapp automatically, and then into X15 form.

```
#ifdef X
package p;
import q.B;
class A { B b; }
#endif
```

(a)

```
package p;
import q.B;

@Feature(X)
class A { B b; }
```

(b)

```
package p;

@Feature(X)
class A { q.B b; }
```

(c)

Figure 15: Translation javapp to @Feature Annotations

In Section 6.1, we said that the four applications in Table 1 used javapp to specify features [28]. In order to use them, we had to reformat javapp to Java custom annotations by hand. We did our best to keep the original feature specification but there were some code fragments that required special care. Example: Fig. 15a shows a compilation unit belonging to optional feature X using javapp. As imports cannot be annotated in Java, we assigned feature X to the class declaration A in Fig. 15b. However, in case class B belongs to X which is unselected, Fig. 15b violates SC: it is an error in Java to import a non-existent class. Our solution was to use the fully qualified name instead as shown in Fig. 15c.

7 RELATED WORK

7.1 A Survey of SPL Tools

Future tools for Java SPLs should have the following properties:

(1) Support the refactoring of SPL codebase $\mathbb{P}$,
(2) Do not create a separate code base for P_C,
(3) Propagate text edits from P_C back to $\mathbb{P}$,
(4) Propagate refactorings of P_C back to $\mathbb{P}$.

because refactorings are central to Java program development; and manual propagation of changes is laborious and error prone [6, 38].

If SPL tools create a separate codebase for P_C, it is possible to automatically propagate edits in P_C to $\mathbb{P}$. But *not* the edits made by refactorings. Why? Recall the Rule of Liebig et al [40]: *An $\mathcal{R}$-refactoring of an SPL fails if $\mathcal{R}$ fails on any product of that SPL.* Refactoring P_C as an isolated codebase will not account for other products of the SPL where that refactoring's precondition fails. Thus, unless a separate codebase for P_C also keeps track of all other products in $\mathbb{P}$, back-propagating of refactorings will fail.

[8] Even variability-aware compilers require source adjustments to be used [34, 40]; there is no free lunch to use any existing SPL tool.

Table 5: Comparing Capabilities of SPL Tooling

Tool	Supports OO refactorings of Java SPLs	Does NOT create separate codebase for products	Back-propagates edits	Back-propagates refactorings
AHEAD	✗	✗	✓	✗
CIDE	✗	✗	✗	✗
DeltaJ	✗	✗	✗	✗
DOPLER	✗	✗	✗	✗
Gears	✗	✗	✓	✗
FeatureHouse	✗	✗	✗	✗
Pure::Variants	✗	✗	✓	✗
X15	✓	✓	✓	✓

Table 5 categorizes the properties of X15 with seven well-known SPL tools (AHEAD [6], CIDE [32], DeltaJ [37], DOPLER [14], Gears [38], FeatureHouse [3], and pure::variants [48][9]). X15 is unique among existing SPL tools in that it supports all key properties.

7.2 Variation Control Systems

Variation Control Systems (VarCSs) are tools that project a reconfigurable codebase $\mathbb{P}$ to produce a separate codebase called a 'view'. The view is edited and its changes are back-propagated to $\mathbb{P}$ by an update tool. AHEAD and Gears, mentioned earlier, are VarCSs [60].

The most advanced VarCSs [57, 60] to our knowledge are based on the Choice Calculus [16] and rely on the *edit isolation principle* (EIP), which says that all edits made to a view are guaranteed not to effect code that was hidden by projection. X15 follows the EIP as long as refactorings are *not* performed; refactorings violate EIP. We showed that propagation tools for text edits are inadequate to deal with the changes refactorings make. Never-the-less, empirical results by Stanciulescu et al. show VarCSs are feasible to edit and maintain real-world SPLs [57]. VarCS ideas offer additional improvements to X15.

7.3 Other Java Variabilities

Consider the Java code of Fig. 16a. Parameter a is Feature-annotated, suggesting that it is removed if X is not a feature of the target configuration. Fig. 16b shows the projected result when ¬X holds. There are SPL tools that support such variability [16, 32, 40].

```
(a) void m( @Feature(X) A a ) {...}    (b) void m() {...}
```

Figure 16: Parameter Removal by Projection

X15 presently ignores Feature annotations on parameters of methods and generics. We are unconvinced that parameter projection is a good idea as it encourages unscalable SPL designs: if method m has 2 parameters in some SPL programs, 3 in others, and 4 in the remainder, it quickly becomes confusing to know which version to use and when. If there are many such methods, an SPL codebase becomes difficult to understand. There is no technical reason that precludes parameter projection in X15 other than increased complexity; we leave its necessity for others to decide and add.

Java annotations have room for improvement. Cazzola et al. [9] presented @Java, an extension to Java language, that can annotate finer-grained code fragments such as blocks and expressions that cannot be annotated by Java. The atjava tool translates @Java annotations to Java-compilable code and then inserts custom attributes into bytecode instead of the translated code. @Java could improve X15 when atjavac (i) provides the start and end of each annotated code fragment, (ii) preserves the original @Java annotation's value expression (i.e., feature expression in X15), and (iii) keeps the annotated expression if it exists. atjavac now supports (i) and (iii), and can be customized to do (ii).

7.4 Variability-Aware Compilers

Conditional compilation in Java has taken two forms: One is OO language-extensions to support type safe variability, such as [4, 15, 27]. These latter papers are elegant proposals to extend OO languages with conditionals to enable static variability and type safety using generics.

The other uses preprocessors, such as [28, 47, 55], which leads to work on VACs [7, 16, 33, 40, 61]. Developing tools to parse C-with-CPP source to analyze the impact of feature variability is difficult [10, 23, 33], but unavoidable if CPP-infused SPL codebases are to be analyzed. Creating a VAC for C++ is far more difficult [11]. Most of the effort in developing VACs deals with the artificial complexity that CPP constructs add to host languages [21, 22]. And using VACs is *not* without effort – the codebase must use disciplined annotations [41].

In contrast to the above research, X15 requires *no changes* to Java or its compiler. X15 directly supports feature-variability for view editing, view compilation, and view refactoring, capabilities that existing SPL tools lack.

7.5 Refactoring Variability-Aware Codebases

Fenske et al. [17] and Schulze et al. [52] report experiences on integrating FeatureHouse [3] with a pair of refactorings: a rename and a pull-up-to-common-feature that partitions large features into a composition of smaller features. Schultz et al. report difficulties on refactoring SPLs when physical feature modularity is used. A deliberate design decision of ours was to use an annotative (or implicit feature modularity) approach to avoid these problems. X15 relies on pure Java, not a custom extension of Java. We argued in Section 4 that theorems of feature algebras hold in all feature-based SPL implementations — including those that rely on special languages to support feature modularity. But to do so requires building a custom compiler and a custom refactoring engine, which is daunting. Never-the-less our results can be transferred to other languages and feature modularity approaches.

There are other useful kinds of feature 'refactoring'. Schulze et al. [53] presented module refactorings such as rename, merge, and remove for SPLs based on *Delta-Oriented Programming (DOP)*. Code smells were proposed to identify refactoring opportunities in DOP [50]. These are potential future extensions of X15.

Kuhlemann et al. [39] proposed *Refactoring Feature Modules* (RFMs). Just as we use the term feature modules to mean building-blocks of SPL products, an RFM is a feature module *or* a single product refactoring (not a refactoring script). An RFM refactoring is feature-unaware and is applied to a feature-unaware product to adapt it for use in a legacy application. Although RFMs have a name that is suggestive of our work, it does not deal with feature-aware refactorings. Nevertheless, subsequently refactoring an SPL program for adaption is a good idea because it separates the concerns for SPL product development and creation from later adaptation.

[9] pure::variants has a tool that updates SPL products when $\mathbb{P}$ is changed [8]; this is forward-propagation $\mathbb{P} \longrightarrow P_C$ of changes, not back-propagation $P_C \longrightarrow \mathbb{P}$ of X15.

Aspect-aware refactorings [1, 25, 43, 62] are a counterpart to feature-aware refactorings. The technical issues and solutions explored were specific to AspectJ (e.g., pointcuts and wild-cards), and are distant topics to OO refactoring feature-based Java SPLs.

8 CONCLUSIONS

Refactoring is a staple of Java software development. It should be a staple of Java SPL development too. X15 is a tool that brings critical OO refactoring support to Java SPL codebases. X15 also avoids two vexing problems: (1) the impossibility of refactoring individual SPL products and correctly propagating changes back to the SPL codebase, and (2) not using a special variability-aware compiler for Java SPL; X15 uses the standard Java compiler. In addition, (i) X15 is a mere 10K Java LOC, (ii) it inherits the benefits of R3: the ability to write and execute refactoring scripts, (iii) efficiently executes scripts (10× faster than the Eclipse JDT refactorings), and (iv) the reason why X15 works – and why any SPL refactoring engine works – is because of a distributivity property of refactorings over feature summations/compositions.

We believe that X15 advances and simplifies the state-of-the-art in SPL tooling.

Acknowledgments. We gratefully acknowledge support for this work by NSF grants CCF-1212683, CCF-1439957 and CCF-1553741.

REFERENCES

[1] P. Anbalagan and T. Xie. Automated Inference of Pointcuts in Aspect-Oriented Refactoring. In *ICSE*, 2007.
[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines.* Springer, 2013.
[3] S. Apel, C. Kästner, and C. Lengauer. FEATUREHOUSE: Language-Independent, Automated Software Composition. In *ICSE*, 2009.
[4] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized Types for Java. In *POPL*, 1997.
[5] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, Sept. 2005.
[6] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, June 2004.
[7] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *ICSE*, 2004.
[8] D. Bueche. Pure::variants functionality. private correspondence, 2016.
[9] W. Cazzola and E. Vacchi. @Java: Bringing a Richer Annotation Model to Java. *Computer Languages, Systems and Structures*, 2014.
[10] S. C. Charles Simonyi, Magnus Christerson. Intentional Software. In *ONWARD! OOPSLA*, 2006.
[11] C++ Parser. http://www.semanticdesigns.com/Products/FrontEnds/CppFrontEnd.html.
[12] K. Czarnecki and K. Pietroszek. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *GPCE*, 2006.
[13] D. Dig and R. Johnson. How Do APIs Evolve&Quest; A Story of Refactoring: Research Articles. *Soft. Maintenance and Evolution: Res. and Pract.*, Mar. 2006.
[14] DOPLER: Decison-Oriented Product LIne Engineering for Effective Reuse. http://ase.jku.at/modules/product-lines/index.html, 2016.
[15] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and Generalized Constraints for C# Generics. In *ECOOP*, 2006.
[16] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM TOSEM*, Dec. 2011.
[17] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *SANER*, 2017.
[18] D. Flannagan. *Java in a Nutshell, 5th Edition.* O'Reilly Publishing, 2005.
[19] R. Fuhrer, A. Kiezun, and M. Keller. Refactoring in the Eclipse JDT: Past, Present, and Future. In *WRT*, volume LNCS 4906, 2007.
[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[21] A. Garrido. *Software Refactoring Applied to C Programming Language.* PhD thesis, University of Illinois at Urbana-Champaign, 2000.
[22] A. Garrido and R. Johnson. Challenges of Refactoring C Programs. In *IWPSE*, 2002.
[23] P. Gazzillo and R. Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *PLDI*, 2012.
[24] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, 1991.
[25] J. Hannemann. Aspect-Oriented Refactoring: Classification and Challenges. In *AOSD*, 2006.
[26] P. Hoefner and B. Moeller. Private correspondence, 2017.
[27] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *AOSD*, 2007.
[28] Java Comment Preprocessor. https://github.com/raydac/java-comment-preprocessor.
[29] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. CMU/SEI-90-TR-021, 1990.
[30] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, University of Magdeburg, 2010.
[31] C. Kästner and S. Apel. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.
[32] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, 2008.
[33] C. Kästner and et al. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*, 2011.
[34] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-aware Module System. In *OOPSLA*, 2012.
[35] J. Kim, D. Batory, and D. Dig. Scripting Parametric Refactorings in Java to Retrofit Design Patterns. In *ICSME*, 2015.
[36] J. Kim, D. Batory, D. Dig, and M. Azanza. Improving Refactoring Speed by 10X. In *ICSE*, 2016.
[37] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. DeltaJ 1.5: Delta-oriented Programming for Java 1.5. In *PPPJ*, 2014.
[38] C. Krueger and et al. BigLever Gears Tool. http://www.biglever.com/solution/product.html.
[39] M. Kuhlemann, D. Batory, and S. Apel. Refactoring Feature Modules. In *ICSR*, 2009.
[40] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware Refactoring in the Wild. In *ICSE*, 2015.
[41] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *AOSD*, 2011.
[42] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the Linux Kernel Variability Model. In *SPLC*, 2010.
[43] M. P. Monteiro and J. a. M. Fernandes. An Illustrative Example of Refactoring Object-oriented Source Code with Aspect-oriented Mechanisms. *Software: Practice and Experience*, April 2008.
[44] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *ICSE*, 2014.
[45] B. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[46] B. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.
[47] Prebop Preprocessor. http://prebop.sourceforge.net/.
[48] pure::variants User Guide. https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf, 2016.
[49] Sat4j. http://www.sat4j.org/.
[50] I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *FOSD*, 2010.
[51] I. Schäfer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *SPLC*, 2010.
[52] S. Schulze, M. Lochau, and S. Brunswig. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *FOSD*, 2013.
[53] S. Schulze, O. Richers, and I. Schaefer. Refactoring Delta-oriented Software Product Lines. In *AOSD*, 2013.
[54] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is The Linux Kernel a Software Product Line? In *SPLC-OSSPL*, 2007.
[55] SLASHDEV Preprocessor. http://www.slashdev.ca/javapp/.
[56] SPL2go. http://spl2go.cs.ovgu.de/.
[57] S. Stanciulescu, T. Berger, E. Walkingshaw, and A. Wasowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *ICSME*, 2016.
[58] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, 2007.
[59] VisualVM 1.3.8. http://visualvm.java.net/.
[60] E. Walkingshaw and K. Ostermann. Projectional Editing of Variational Software. In *GPCE*, 2014.
[61] L. Wanner, L. Lai, A. Rahimi, M. Gottscho, P. Mercati, C.-H. Huang, F. Sala, Y. Agarwal, L. Dolecek, N. Dutt, P. Gupta, R. Gupta, R. Jhala, R. Kumar, S. Lerner, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, D. Sylvester, and Y. Zhou. NSF Expedition on Variability-Aware Software: Recent Results and Contributions. *Information Technology*, 2015.
[62] J. Wloka, R. Hirschfeld, and J. Hänsel. Tool-supported Refactoring of Aspect-oriented Programs. In *AOSD*, 2008.