Refactoring Java Software Product Lines

Jongwook Kim  
University of Texas at Austin  
jongwook@cs.utexas.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 a 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.

1 INTRODUCTION

Refactoring has been a staple in Object-Oriented (OO) programming for at least a quarter century [25, 46], and a standard tool in Integrated Development Environments (IDEs) for at least a decade [20]. Software Product Lines (SPLs) have an equally long and rich history [2, 30]. Despite this progress, there are no tools – research prototypes [3, 8, 32, 37] or commercial tools [16, 38, 49] – 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, 55, 58]. 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].

Variability in a SPL codebase relies on presence conditions, a predicate expressed in terms of features, that indicate when a fragment of code, file or package is to be included in an SPL product. A typical use-case is with #if-#endif presence condition of typical use-case is with #if-#endif of code, file or package is to be included in an SPL product. A feature predicate to remove unreachable code in if(feature_exp) statements. But removing entire declarations (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 [29, 48, 56], although officially Java shuns preprocessors [19]. Another way is to copy and assemble code fragments from an SPL codebase P to produce an SPL product P_C where C is C_S configuration [3, 8, 32, 38]. Both create a separate codebase for P_C that a user edits to improve, tune, and repair P_C. Doing so exposes two critical problems in SPL tooling. First, given an edited product P_C, how are its edits propagated back to P, the SPL codebase? Early SPL tools [8, 38] had back-propagation capabilities. Further, there are many prototype tools for projecting CPP codebases to ‘view’ codebases that can be edited and their changes back-propagated to P (see [59, 62] for surveys). But none correctly propagates changes from P_C to P made by refactoring. Why? Renaming a field in P_C is easy, but not all references of the field reside in P_C; other references may exist in P that are not in P_C. Thus, back-propagating edits will rename some, but not all, references to a field, breaking P. In short, SPL back-propagation tools must become ‘refactoring-aware’.

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 – is the current solution to this impasse [9, 18, 33, 63]. VACs generate AST nodes with presence conditions and variability-aware control-flow graphs needed for both precondition checks and code transformations of SPL refactorings [40]. But writing a VAC – even for the C-language – is daunting [13, 24, 33]. We are unaware of any VAC for a mainstream OO language. And even if such a VAC existed, 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 P.

The novel contributions of this paper are:

- The X15 tool for editing, projecting, and refactoring Java SPLs and their products;
- Identifying primitive refactoring 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].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC ’17, Seville, Spain  
© 2017 ACM. ISBN: 978-1-4503-XXXX-X/YY/MM. . $15.00  
DOI: 10.1145/xxxxx.xxxxxx

2 THE R3 REFACTORING ENGINE

R3 is a state-of-the-art refactoring engine for Java [36]. It is an improvement over the Eclipse Java Development Tools (JDT) refactoring engine as it allows programmers to write refactoring scripts — programmatic sequences of refactoring invocations — and that it executes these scripts 10X 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. Information on these objects, like the presence of a private modifier, can be harvested via 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 Java class refactorings (e.g., rename or move) or methods to find related objects (e.g., RField and RMethod objects of that RClass). This enables many Gang-of-Four Design Patterns [21] to be partially or fully automated, written as simple Java methods [35, 36].

Consider the adapter pattern. An Adapter is a class that implements the public methods of a Target class by invoking methods of an Adaptee class. The R3 makeAdapter method of Fig. 1 works by (1) retrieving the package of the Target class, (2) creating a class named adapterClassName as a subclass of Target in this package, (3) creating a field named ‘adapterFieldName’ of type Adaptee, (4) creating a constructor with an argument that initializes this field, and (5) for each public and non-static methods of Target, creating a method stub (which has programmer /X TO D05*/ for that method), and (6) returning the created adapter to the makeAdapter caller. Because there are /X TO D05*/ , the adapter pattern is partially automatable [35, 36].

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].

X15 uses Java’s built-in annotations to encode variability; we will shortly explain how X15 realizes variabilities. Let P denote the code base of a Java SPL. Every Java declaration (class, method, field, constructor, initializer) in P has an optional Feature annotation with a boolean expression of Feature variables. 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.

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 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.

---

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

---

Figure 1: An R3 makeAdapter Method.

Figure 2: The Feature Annotation Type

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 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.
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 made directly to P. End of story.

The problem is that refactorings are more than text edits. Consider the SPL codebase 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 P breaks P because not all occurrences of Grafix in P are renamed to Graphics — the program for configuration (Y) no longer compiles.

In a nutshell, text-edit back-propagation tools for SPLs are not ‘refactoring-aware’; they are inadequate to deal with the changes made by refactorings. Dig and Johnson demonstrated an analogous problem for version control [15]. In effect, future SPL tools must provide ‘refactoring aware’ back-propagation. The key insight to achieve ‘refactoring awareness’ is discussed next.

4 ALGEBRAS OF FEATURE COMPOSITIONS

Features have long been viewed as the conceptual modules or building blocks of SPL products. Early research (AHEAD [8], FeatureHouse [3], DOP [51]) not only developed algebras for feature compositions, but also invented OO language extensions to define concrete feature modules. While the ideas behind these language extensions — role-based programming, mixin-layers, and context-oriented programming — have been widely explored, we sketch known ideas and then present the insight that made X15 possible.
4.1 Sum and Projection of Feature Modules

Algebras axiomatize the summation or composition of feature modules to produce SPL products [6, 7]. They abstract and unify concepts of disparate SPL implementations [2, 31]. For example in annotative approaches, code fragments of a feature are distributed throughout codebase \( P \). At another extreme, a feature module collects these same fragments in a single package-like structure. Any theorem that can be proven using feature modules should hold for all feature-based SPL implementations.

A feature module \( F_i \) encapsulates the implementation of feature \( i \). Product \( P_c \) with configuration \( C \) is produced by summing the modules of its features \( \{X, Y, Z\} \) where \( X, Y \), and \( Z \) are features, product \( P_c \) is:

\[
P_C = \sum_{i \in C} F_i = F_X + F_Y + F_Z
\]

As said in Section 3, \( X_{15} \). Product \( P_C \) is produced by summing the modules of its features \( \{X, Y, Z\} \) where \( X, Y \), and \( Z \) are features, product \( P_c \) is:

\[
P_C = \sum_{i \in C} F_i = F_X + F_Y + F_Z
\]

Let \( F \) be the set of all features. The codebase \( P \) of an SPL is:

\[
P = \sum_{i \in \text{all features}} F_i
\]

Projection, as discussed in Section 3, is a complementary operation to summation. The \( C \)-projection of \( P \) yields \( P_c \):

\[
\Pi_C(F) = P_c
\]

Think of projection as the operation that eliminates feature modules that do not belong to \( C \). Let \( C_1 \) and \( C_2 \) be different sets of features from the same SPL (i.e., \( C_1, C_2 \subseteq F \)). An axiom that relates projection and summation is:

\[
\Pi_{C_1}(\sum_{i \in C_2} F_i) = \sum_{i \in C_1 \cap C_2} F_i
\]

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

\[
\Pi_C(F) = \Pi_C(\sum_{i \in F} F_i) \quad \text{// (2)}
\]

\[
= \sum_{i \in C \cap F} F_i \quad \text{// (4)}
\]

\[
= \sum_{i \in C} F_i \quad \text{where } C \subseteq F
\]

As said in Section 3, X₁₅ implements projection in two different ways: \( \Pi_{\text{old}} \) code-folds \( P \) to expose only the code of \( P_c \) for viewing, editing and refactoring. \( \Pi_{\text{new}} \) comments-out unnecessary code which is then fed to the Java compiler to produce bytecodes for \( P_c \); this compiled version enables programmers to execute, debug, and step-through the code folded version of \( P_c \).

4.2 Theorem for Refactoring SPLs

The unknown is this: how do refactorings extend the algebras of feature compositions? No one to our knowledge has answered this question before; an answer will tell us how SPL codebases can be refactored.

Let \( R \) be a refactoring. If we \( R \)-refactor \( P_c \), we get \( P^R_c \):

\[
\mathcal{R}(\Pi_C(F)) = \mathcal{R}(P_C) = P^R_C
\]

As \( R \) changes \( P_c \), \( R \) must also change \( P \). But how? Our conjecture and theorem is this: \( P^R_c \) can be computed by the \( R \)-refactoring of \( P \) followed by a \( C \)-projection:

\[
\Pi_C(\mathcal{R}(P)) = P^R_C
\]

Equivalently, (6) is the commuting diagram of Fig. 8 where the operations of projection and refactoring commute [47].

SPL programmers must realize that refactoring an SPL codebase \( P \) has more constraints than just refactoring a single product \( P_c \). We explain in Section 5.2 that the preconditions to \( R \)-refactor \( P \) imply the preconditions to \( R \)-refactor \( P_c \).

Our proof of (6) assumes the preconditions to \( R \)-refactor \( P \) are satisfied. Therefore \( R \) in (6) really represents the code transformation that is made by an \( R \)-refactoring.

We observed the following distributivity identity over years of developing feature-based SPLs: the \( R \)-refactoring of a sum of feature modules \( A \) and \( B \) equals the sum of the each \( R \)-refactored feature module:

\[
\mathcal{R}(A + B) = \mathcal{R}(A) + \mathcal{R}(B)
\]

This axiom is intuitive: \textit{common refactorings are largely oblivious to feature module boundaries}. That is, when a program \( P = A + B \) is \( R \)-refactored, one expects both modules \( A \) and \( B \) to be modified by \( R \), namely \( P^R = A^R + B^R \).

\textbf{Example: } Method \texttt{m} in Fig. 9 is defined in class \texttt{A}. Class/feature \texttt{B} calls \texttt{m}. When \texttt{m} is renamed to \( n \), both features \( A \) and \( B \) are modified to \( A^R \) and \( B^R \).

![Figure 8: Theorem of SPL Refactoring](image)

![Figure 9: Rename-Method Refactoring](image)

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

\[
\Pi_C(\mathcal{R}(P)) = \Pi_C(\mathcal{R}(\sum_{i \in F} F_i)) \quad \text{// by (2)}
\]

\[
= \Pi_C(\mathcal{R}(\sum_{i \in C} F_i)) \quad \text{// by (7)}
\]

\[
= \sum_{i \in C} \mathcal{R}(F_i) \quad \text{// by (4)}
\]

\[
= \mathcal{R}(\sum_{i \in C} F_i) \quad \text{// by (7)}
\]

\[
= \mathcal{R}(P_C) \quad \text{// by (5)}
\]

\[
= P^R_C \quad \text{// by (5)}
\]

\footnote{A cross-product of features exposes the submodules of features that arise from feature interactions [6, 54]. Cross-products rely on module summation, and are otherwise orthogonal to this paper.}

\footnote{A common name for \( F \) is a 150% design – it includes all possibilities.}
Equation (6) tells us how to translate refactorings of SPL products, namely refactorings to $P_C$, to refactorings of the SPL codebase $P$.

When an SPL programmer applies a refactoring $R$ on of $P_C$, s/he sees $R(P_C) = P_C^R$ as the result. But behind the curtains, $X_{15}$ is really applying $R$ to $P$, and taking its $C$-projection to present $P_C^R$ to the programmer.\footnote{Equation (6) also tells us that algebras for feature summation and refactoring are elegant. The name of this algebraic structure is a 'left $N$-semimodule over a monoid' [27].}

**Example:** A $X_{15}$ user renames Grafix to Graphics in $P_X$ of Fig. 7. $X_{15}$ applies this refactoring to the entire codebase $P$. 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. $X_{15}$ updates all programs in an SPL that are affected by this rename, and thus keeps $P$ consistent.

## 5 Refactoring Preconditions

Applying a code transformation $R$ to a codebase is well-understood [10, 36]. An interesting part about $X_{15}$ is how it handles 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, 14, 33, 34, 45, 60].

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 $\psi := (F \Rightarrow X \land Y)$. That is, when statement "$x = y$" appears in a program, so must the declarations for $x$ and $y$.

Let $\phi$ be the propositional formula of the SPL’s FM [2, 5]. If $\phi \land \lnot \psi$ is satisfiable, then at least one program in the SPL does not satisfy $\psi$ and hence will not compile [14]. Similarly, **dead code** is source that appears in no SPL program. Let $\delta$ be the presence condition for code fragment $\ell$. If $\phi \land \lnot \psi$ is unsatisfiable, then $\ell$ is dead code.

An SC tool calls $P$ for all distinct $\psi$ and $\delta$ and verifies that no program in the SPL violates either constraint. We say $P$ satisfies SC if no presence condition $\psi$ is violated and $P$ is **dead code free** if no dead code fragments are found.

### 5.2 Preconditions for SPL Refactorings

Theorem (6) assumes the preconditions for $R$-refactoring $P$ are satisfied. But what are these preconditions? Consider this example: A programmer wants to refactor the base product $P_{base}$ whose SPL codebase $P$ 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 $R$-refactoring of an SPL fails if $R$ fails on any product of that SPL.

$X_{15}$ reports precondition failures of a refactoring $R$ by citing a condition or SPL configuration where it fails. This is done by ‘lifting’ a refactoring precondition to a SC constraint $\psi$ and verifying all SPL products satisfy $\psi$. (By definition the lifted constraint implies the precondition on program $P_C$). $R_3$ supports 39 different primitive refactorings and uses 39 distinct primitive precondition checks, where each $R_3$ refactoring uses a subset of these 39 checks. $X_{15}$ supports all of $R_3$’s primitive refactorings and preconditions.

We expected most $R_3$ 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 $R_3$ counterparts. And

(2) Some preconditions are feature-independent, such as Declaring Type\footnote{A method cannot be moved if its enclosing type is an annotation or interface.} and Constructor,\footnote{A constructor cannot be moved.} so lifting them is unnecessary.

Here are the preconditions that required lifting:

- **Binding Resolution.** Before a method is moved, a lifted check is performed: the moved method should still be present in all programs in which it appeared before the move and all declarations referenced in its body are still present and visible, otherwise the move refactoring is rejected. Fig. 11 shows the before and after result of moving method $A.m$ to class $C$. One SC check for parameter type $B$ prior to the move is $GREEN \land BLUE \Rightarrow YELLOW\footnote{Also, the presence of method $n$ implies the presence of class $C$.}$ and after the move the check becomes $RED \land BLUE \Rightarrow YELLOW$.

![Figure 11: Binding Resolution Constraint](image-url)
will a SAT solver be invoked, and of course, its result is henceforth cached. The cache is cleared whenever the feature model is updated.

A saving grace is that the number of unique SAT checks is small, possibly orders of magnitude smaller than the crude estimate \(\frac{1}{2}\) SAT check per every 2 lines of source. From experimental results in Section 6, a crude estimate is about 1 SAT check per every 2 lines of source.

5.3 Implementation Notes
Feature models of SPLs are rather static; they do change but slowly. X15 culls \(\mathcal{P}\) 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 [60].

X15 leverages the stability of an SPL’s feature model by caching the results of SAT checks. 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 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 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 [21]; the other 5 patterns do not benefit from automation [56].

We answer three research questions:
- \(\text{RQ1:}\) Can X15 refactor Java SPLs?
- \(\text{RQ2:}\) How fast is X15 compared to R3?
- \(\text{RQ3:}\) 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 [57]. Column \textbf{App} of Table 1 lists the eight target SPLs along with their lines of code, number of regression tests, and number of features. Three SPLs (AHEAD, Calculator, and Elevator) had regression tests that could validate X15 transformations worked correctly. Two (Notepad and Sudoku) 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 (Lampiro, MobileMedia, and Preyavler) 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.

We used an Intel CPU i7-2600 3.40GHz, 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) [61]. Each experiment was executed five times and the average is reported.

6.2 Results
6.2.1 Table Organization. Table 1 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 [21, 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 number (M#). The third 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 \textbf{R3 Time} (R3T).
The next three lists number 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 precondition checks (with/without SAT-caching), including SAT-solving to check feature-aware refactorings and the time when 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 2 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, A5, executed 552 primitive refactorings. Other experiments required fewer as their visitor class had fewer ‘visit’ methods.

### RQ2: How fast is X15 compared to 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 A5, took 4.77 seconds. 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 A5 took Eclipse 298 seconds to execute, a speedup of over 100× [36].)

2. Row L5 took 5.44 seconds; the comparable experiment using R3 took 3.62 seconds. The numbers for inverseVisitor in Table 2 are similar. For less demanding scripts — remember: rows are not individual refactorings — all 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.

3. X15 harvests information from P before it can execute a script. X15 must collect more information; specifically feature presence predicates (see column α of Table 1). 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.

Towards O(8%) of Eclipse refactoring execution time is consumed by checking predicates [36]. In contrast, R3 precondition checking is almost instantaneous [36]. X15 takes advantage
of R3’s speed, but spends extra time for feature-aware pre-condition 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 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 L5 spent 1.16 seconds proving 1,294 theorems, a vast majority of which were duplicates. With caching, only one extra theorem required a SAT proof, taking 0.06 seconds.

(2) Table 3 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×.

Table 3: Dead Code and Safe Composition Check Results

<table>
<thead>
<tr>
<th>App</th>
<th>No-caching (seconds)</th>
<th>Caching (seconds)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>SC</td>
<td>DC</td>
<td>SC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.34 [176]</td>
<td>0.95 [15]</td>
</tr>
<tr>
<td>B</td>
<td>0.22 [185]</td>
<td>0.36 [676]</td>
<td>0.23 [155]</td>
</tr>
<tr>
<td>C</td>
<td>0.23 [185]</td>
<td>0.36 [676]</td>
<td>0.23 [155]</td>
</tr>
<tr>
<td>D</td>
<td>0.22 [185]</td>
<td>0.36 [676]</td>
<td>0.23 [155]</td>
</tr>
<tr>
<td>E</td>
<td>0.22 [185]</td>
<td>0.36 [676]</td>
<td>0.23 [155]</td>
</tr>
<tr>
<td>F</td>
<td>0.22 [185]</td>
<td>0.36 [676]</td>
<td>0.23 [155]</td>
</tr>
</tbody>
</table>

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

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 due to the fact that the feature models of our SPLs are rather simple. Having said this, our observations are consistent with prior work that SAT problems for feature models are ‘easy’ [43].

Our conclusion: caching solutions to SAT checks does indeed improve performance.

6.3 Converting SPL Codebases to X15 Format
Every SPL tool today uses a unique means to encode variability. In order to use these SPLs in our experiments, we had to modify them to use X15 annotations.

SPLs that used AHEAD [8] and FeatureHouse [3], namely Mixin, Calcutor and Elevator, were partially translated by tools – manual work was still needed. The remaining five applications (Notepad, Sudoku, Lampiro, MobileMedia, and Prevayler) used CIDE [32].

which could be transformed into javapp automatically, and then into X15 form.

![Figure 15: Translation javapp to @Feature Annotations](image)

In Section 6.1, we said that the four applications in Table 1 used javapp to specify features [29]. 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 import's 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 P,
(2) Do not create a separate code base for PC,
(3) Propagate text edits from PC back to P,
(4) Propagate refactorings of PC back to P.

because refactorings are central to Java program development; and manual propagation of changes is laborious and error prone [8, 38].

If SPL tools create a separate codebase for PC, it is possible to automatically propagate edits in PC to P. But not the edits made by refactorings. Why? Recall the Rule of Liebig et al. [40]: An R-refactoring of an SPL fails if R fails on any product of that SPL. Refactoring PC 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 PC also keeps track of all other products in P, back-propagating of refactorings will fail.

Table 4 categorizes the properties of X15 with eight well-known SPL tools (AHEAD [8], CIDE [32], Delta [37], DOPLER [16], Gears [38], FeatureHouse [3], and pure::variants [49]). X15 is unique among existing SPL tools in that it supports all key properties.

Table 4: Comparing Capabilities of SPL Tooling

<table>
<thead>
<tr>
<th>Tool</th>
<th>Supports OS refactorings of Java SPLs</th>
<th>Does NOT create separate codebase for products</th>
<th>Back-propagates edits</th>
<th>Back-propagates refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHEAD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CIDK</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Delta</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DOPLER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Gears</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FeatureHouse</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Pure-Variants</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X15</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
7.2 Variation Control Systems

Variation Control Systems (VarCSs) are tools that project a reconfigurable codebase $P$ to produce a separate codebase called a ‘view’. The view is edited and its changes are back-propagated to $P$ by an update tool. AHEAD and Gears, mentioned earlier, are VarCSs [62]. The most advanced VarCS tools [59, 62] are based on the Choice Calculus [18] 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 Stanculescu et al. show VarCS tools are feasible to edit and maintain real-world SPLs [59]. 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 [18, 32, 40].

(a) \texttt{void m( @Feature(X) A a ) {...}}

(b) \texttt{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 unscaleable 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 impossible 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. [12] 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 \texttt{atjava} tool translates @Java annotations to Java-composable 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, 17, 28]. 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 [29, 48, 56], which leads to work on VACs [9, 18, 33, 40, 63]. Developing tools to parse C-with-CPP source to analyze the impact of feature variability is difficult [13, 24, 33], but unavoidable if CPP-infused SPL codebases are to be analyzed. It may be years, if ever, before a VAC for C++ appears. Most of the effort in developing VACs deals with the artificial complexity that CPP constructs add to host languages [22, 23]. And using these 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

Schulze et al. [32] report experiences on integrating FeatureHouse [3] with refactorings, such as pull-up, but also refactorings that partition large features into a composition of smaller features. The authors 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 that the mathematics of (6) applies to 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.

There are other useful kinds of feature ‘refactoring’. Schulze et al. [53] presented module refactorings such as rename, merge, and remove for DOP SPLs. 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, subsequent refactoring an SPL program for adaption is a good idea because it separates the concerns for SPL product development and creation from later adaptation.

Aspect-aware refactorings [1, 26, 44, 64] 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 program development. It should be a staple of Java SPL development too. X15 is a tool that not only brings critical refactoring support to Java SPLs, it also solves three other vexing problems: (1) propagation of edits and refactorings of SPL products back to the SPL codebase, (2) not requiring language extensions to Java or a special variability-aware compiler; X15 uses the standard Java compiler and (3) efficiency: in running the most demanding refactoring scripts that execute over 558 individual OO refactorings, X15 runs but a few seconds slower than R3, its feature-unaware refactoring engine counterpart. In addition, (i) X15 is a
meme 10k Java LOC, (ii) it inherits the benefits of R3: the ability to write and execute refactoring scripts, and (iii) efficiently executes scripts (10x faster than the Eclipse JDT refactoring).

X15 was inspired by abstractions of feature composition. What was unknown prior to this work was the relationship between the summation (composition) of feature modules and refactorings. In years of work developing feature-based SPLs, we observed the following distributivity axiom: The refactoring of a sum of feature modules is the sum of refactored feature modules. The success of X15 to refactor SPLs demonstrates the correctness of this axiom and practicality of feature abstractions.

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

Acknowledgments. We thank Denis Bogdanas and the referees for their valuable comments on an early draft of this paper. We gratefully acknowledge support for this work by NSF grants CCF-1212683, CCF-1439957 and CCF-1553741.

REFERENCES

For their valuable comments on an early draft of this paper. We

63. L. Wanner and et al. NSF Expedition on Variability-Aware Software: Recent Results and Contributions. Information Technology, 2015.

Jongwook Kim, Don Batory, and Danny Dig