Semantic Patches for Java Program Transformation

Hong Jin Kang
School of Information Systems, Singapore Management University, Singapore

Ferdian Thung
School of Information Systems, Singapore Management University, Singapore

Julia Lawall
Sorbonne Université/Inria/LIP6, France

Gilles Muller
Sorbonne Université/Inria/LIP6, France

Lingxiao Jiang
School of Information Systems, Singapore Management University, Singapore

David Lo
School of Information Systems, Singapore Management University, Singapore

Abstract

Developing software often requires code changes that are widespread and applied to multiple locations. There are tools for Java that allow developers to specify patterns for program matching and source-to-source transformation. However, to our knowledge, none allows for transforming code based on its control-flow context. We prototype Coccinelle4J, an extension to Coccinelle, which is a program transformation tool designed for widespread changes in C code, in order to work on Java source code. We adapt Coccinelle to be able to apply scripts written in the Semantic Patch Language (SmPL), a language provided by Coccinelle, to Java source files. As a case study, we demonstrate the utility of Coccinelle4J with the task of API migration. We show 6 semantic patches to migrate from deprecated Android API methods on several open source Android projects. We describe how SmPL can be used to express several API migrations and justify several of our design decisions.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Program transformation, Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.13

Category Experience Report

Supplement Material Coccinelle4J can be found at https://github.com/kanghj/coccinelle/tree/java

Funding This research was supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and the ANR ITrans project.

1 Introduction

Over ten years ago, Coccinelle was introduced to the systems and Linux kernel developer communities as a tool for automating large-scale changes in C software [23]. Coccinelle particularly targeted so-called *collateral evolutions*, in which a change to a library interface triggers the need for changes in all of the clients of that interface. A goal in the development of Coccinelle was that it should be able to be used directly by Linux kernel developers, based on their existing experience with the source code. Accordingly, Coccinelle provides a language, the Semantic Patch Language (SmPL), for expressing transformations using a generalization of the familiar patch syntax. Like a traditional patch, a Coccinelle semantic patch consists of fragments of C source code, in which lines to remove are annotated with -

and lines to add are annotated with +. Connections between code fragments that should be executed within the same control-flow path are expressed using "..." and arbitrary subterms are expressed using metavariables, raising the level of abstraction so as to allow a single semantic patch to update complex library usages across a code base. This user-friendly transformation specification notation, which does not require users to know about typical program manipulation internals such as abstract-syntax trees and control-flow graphs, has led to Coccinelle's wide adoption by Linux kernel developers, with over 6000 commits to the Linux kernel mentioning use of Coccinelle [14]. Coccinelle is also regularly used by developers of other C software, such as the Windows emulator Wine[1] and the Internet of Things operating system Zephyr.[2]

Software developers who learn about Coccinelle regularly ask whether such a tool exists for languages other than C.[3] To begin to explore the applicability of the Coccinelle approach to specifying transformations to software written in other languages, we have extended the implementation of Coccinelle to support matching and transformation of Java code. Java is currently the most popular programming language according to the TIOBE index.[4] The problem of library changes and library migration has also been documented for Java software [29]. While tools have been developed to automate transformations of Java code [20, 25, 28], none strikes the same balance of closeness to the source language and ease of reasoning about control flow, as provided by Coccinelle. Still, Java offers features that are not found in C, such as exceptions and subtyping. Thus, we believe that Java is an interesting target for understanding the generalizability of the Coccinelle approach, and that an extension of Coccinelle to Java can have a significant practical impact.

Our research goal is to explore what can be done for Java programs with the Coccinelle approach, i.e. transformation rules expressed in a patch-like notation and other features from Coccinelle. This experience paper documents our design decisions for Coccinelle4J, our extension of Coccinelle to handle Java code. We present the challenges we have encountered in the design of Coccinelle4J and the initial implementation. The design has been guided by a study of the changes found in the development history of five well-known open-source Java projects. Through a case study where we transform deprecated API call sites to use replacement API methods, we evaluate Coccinelle4J in terms of its expressiveness and its suitability for use on Java projects.

This paper makes the following contributions:

- We show that the approach of program transformation of Coccinelle, previously only used for C programs, generalizes to Java programs.
- We document the design decisions made to extend Coccinelle to work with Java.
- In the context of migrating APIs, we use Coccinelle4J and show that control-flow information is useful for Java program transformation.

The rest of this paper is organized as follows. Section 2 briefly introduces Coccinelle. Section 3 presents our extensions to support Java. Section 4 looks at a case study involving migrating APIs in open source Java projects. Section 5 discusses related work. Finally, we present our conclusions and discuss possible future directions of this work in Section 6.

[1] `https://wiki.winehq.org/Static_Analysis#Coccinelle`
[2] `https://docs.zephyrproject.org/latest/application/coccinelle.html`
[3] https://twitter.com/josh_triplett/status/994753065478582272
[4] `https://www.tiobe.com/tiobe-index/`, visited January 2019

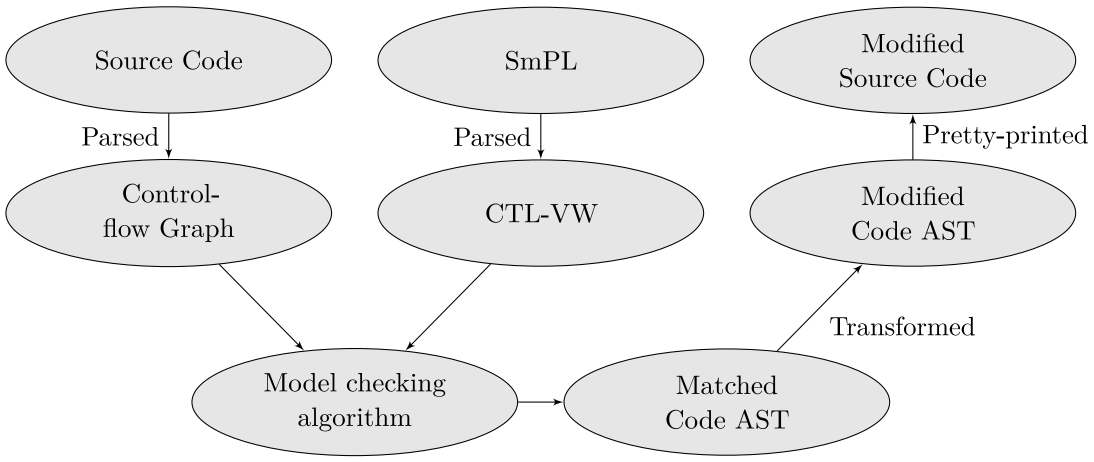

Figure 1 The process of transforming programs for a single rule

2 Background

Coccinelle is a program transformation tool with the objective of specifying operating system collateral evolutions [19], code modifications required due to changes in interfaces of the operating system kernel or driver support libraries. Similar to widespread changes, such an evolution may involve code that is present in many locations within a software project. It was found that, to automate such changes, it is often necessary to take into account control-flow information, as collateral evolutions may involve error-handling code after invoking functions, or adding arguments based on context [3]. Much of the design of Coccinelle has been based on pragmatism, trading off correctness for ease of use and expressiveness. Coccinelle works on the control-flow graph and Abstract Syntax Tree (AST) for program matching and transformation, and therefore matches on program elements regardless of variations in formatting.

Coccinelle's engine is designed based on model checking. Building on an earlier work by Lacey and De Moor [12] on using Computational Tree Logic (CTL) to reason about compiler optimizations, Brunel et al. [3] propose CTL-VW as a foundation for program matching on control-flow graphs in Coccinelle. CTL-VW extends CTL, by adding predicates over metavariables that can be existentially qualified over program elements, as well as by adding witnesses, which record variable bindings and track locations to be transformed.

As input, Coccinelle accepts source code and a semantic patch describing the program transformation. The source code is parsed, producing an intraprocedural control-flow graph for each function, while the semantic patch language is converted into a CTL-VW formula. For each function definition or other top-level declaration in the source code, the CTL-VW formula is matched against the control-flow graph using a model-checking algorithm. This process is summarized in Figure 1.

Semantic patches are specified in SmPL, which has a declarative syntax resembling patches produced by the familiar `diff` command. A semantic patch describes code that is removed and added, with any relevant surrounding code given as context. '`-`' and '`+`' are used to mark lines containing code to be removed and added, respectively. This syntax follows a WYSIWYG approach and is familiar to developers using revision control software. Listing 1 shows a simplified version of a semantic patch checked into the mainline Linux kernel repository. This semantic patch removes declarations of variables that are assigned a constant value and immediately returned. It makes use of some notation unique to Coccinelle that we will explain further below.

Listing 1 Example of a rule that removes variables that only exist to return a constant

```
1  @@
2  type T;
3  constant C;
4  identifier ret;
5  @@
6  - T ret = C;
7  ... when != ret
8      when strict
9  return
10 - ret
11 + C
12 ;
```

We briefly describe semantic patches and SmPL, using Listing 1 as an example. A rule in a semantic patch has two parts: the declaration of metavariables (lines 2-4), followed by the specification of the intended transformation over the declared metavariables (lines 6-12). Although our example contains only a single rule, a semantic patch can consist of multiple rules. Rules after the first one can refer to metavariables matched by prior rules.

Coccinelle offers abstractions to reason about paths. The "`...`" operator in SmPL represents an arbitrary sequence. In the context of Listing 1, the operator represents any sequence of statements over any control flow path. The `when` keyword restricts matches of "`...`". In Listing 1, the use of `when != ret` means that there should be no occurrences of `ret` in the matched control-flow path, and the use of `when strict` means that this property applies to all control-flow paths starting from the declaration, including those that abort with an error value. Without `when strict`, all paths must be matched *except* those that end with an error abort. A further option is `when exists`, which is satisfied whenever a single path meets the specified criteria on any other `when` constraints. By default, a control-flow path matched with "`...`" may not contain code that matches the patterns preceding or following the "`...`", here the declaration of `ret` and the return of its value, thus matching the shortest path between these points. This constraint can be lifted using `when any`. One caveat is that Coccinelle does not account for the run-time values when computing paths, and thus over-approximates the set of paths that can be taken.

As we do not modify the semantics of SmPL, we do not discuss SmPL in further detail. A more complete treatment of SmPL is given by Padioleau et al. [24].

3 Extending Coccinelle to Java

In this section, we document our experience and design decisions made during the development of Coccinelle4J. Our first observation is that much of the syntax of Java is also found in C. For example, both languages contain `if` statements, assignments, function/method calls, etc. And even some differences in the syntax, such as the ability to declare a variable in the header of a `for` loop, amount to only minor variations. Thus, we can use much of the implementation of Coccinelle unchanged and provide a transformation language that has a similar look and feel.

Table 1 Number of commits for each type of change

Type of change	Number of commits
Addition of new methods or modification of method signatures	43
Modification of multiple methods' implementation	33
Changes that are non-functional	23
Modification of imports	14
Changes related to sub-typing or inheritance	9
Modification of annotations	8

Still, Java is a different language than C, and notably offers an object-oriented programming paradigm, rather than a simple imperative one, as found in C. To identify the language features that are most relevant to widespread changes in Java code, we performed an analysis based on commits in the past year of five common Java libraries: Gson,[5] Apache Commons IO[6] and Commons Lang,[7] Mockito,[8] and Selenium.[9] Out of the 1179 commits to these projects, we found 130 containing a widespread change, i.e., a change that is both semantically and syntactically similar made in multiple locations. Depending on the widespread change present in the commit, we manually categorized these commits as shown in Table 1. We use the frequency of each category of change to motivate and prioritize the necessary features of Coccinelle that we port to Coccinelle4J. In particular, we notice that the features of class hierarchy and annotations are changed least frequently, so these features are not offered in the current design of Coccinelle4J.

Guided by this analysis, we have developed Coccinelle4J in three phases. As a first milestone for our work, we target the constructs found in Middleweight Java [1]. Middleweight Java is a minimal imperative fragment for Java, designed for the study of the Java type system. In contrast to other formal models of Java, Middleweight Java programs are valid executable Java programs, thus representing a useful first step for our work. Still, most real Java programs involve constructs that go beyond the very limited syntax of Middleweight Java. With ad-hoc testing on some real projects, we identified the need for handling control-flow with exceptions in the context of `try-catch`, and the need to introduce Java-specific *isomorphisms*, Coccinelle meta-rules that make it possible to match semantically equivalent code with concise patterns. These features were added in the second phase. Finally, our third phase introduces the ability to reason to a limited degree about subtyping. Across the three phases, we added or modified a total of 3084 lines of code.

Much like the original design of Coccinelle, we focus on pragmatism and provide most features on a best-effort basis, without proving correctness. Like Coccinelle, Coccinelle4J does not crash or throw any errors when it encounters code that it cannot parse or transform. Instead, it recovers from any errors and ignores the parts of the code that it cannot handle.

3.1 Phase 1: Middleweight Java

Middleweight Java is a minimal but valid subset of Java that still retains features of Java such as field assignments, null pointers, and constructor methods. The syntax of Middleweight

[5] `https://github.com/google/gson`
[6] `https://gitbox.apache.org/repos/asf?p=commons-io.git`
[7] `https://gitbox.apache.org/repos/asf?p=commons-lang.git`
[8] `https://github.com/mockito/mockito`
[9] `https://github.com/SeleniumHQ/selenium`

Java programs is as follows:

Program

$$p ::= cd_1 ... cd_n; \bar{s}$$

Class definition

$$\begin{aligned} cd ::= {} & \texttt{class}\ C\ \texttt{extends}\ C\ \texttt{\{} \\ & \quad fd_1 ... fd_k \\ & \quad cnd \\ & \quad md_1 ... md_n \\ & \texttt{\}} \end{aligned}$$

Field definition

$$fd ::= C f;$$

Constructor definition

$$\begin{aligned} cnd ::= {} & C(C_1 x_1, ..., C_j x_j)\ \texttt{\{} \\ & \quad \texttt{super(}e_1, ..., e_k\texttt{)}; s_1 ... s_n \\ & \texttt{\}} \end{aligned}$$

Method definition

$$md ::= \tau\ m(C_1 x_1, ..., C_n x_n)\ \texttt{\{}s_1 ... s_k\texttt{\}}$$

Return type

$$\tau ::= C \mid \texttt{void}$$

Expression

$e ::= x$	*Variable*
$\mid \texttt{null}$	*Null*
$\mid e\texttt{.}f$	*Field access*
$\mid \texttt{(}C\texttt{)}e$	*Cast*
$\mid pe$	*Promotable expression*

Promotable Expression

$pe ::= e.m(e_1, ..., e_k)$	*Method invocation*
$\mid \texttt{new}\ C\texttt{(}e_1, ... e_k\texttt{)}$	*Object creation*

Statement

$s ::= \texttt{;}$	*No-op*
$\mid pe;$	*Promoted expression*
$\mid \texttt{if (}e\texttt{==}e\texttt{)}\ s_1 ... s_k\ \texttt{else}\ s_{(k+1)} ... s_n$	*Conditional*
$\mid e\texttt{.}f\texttt{=}e;$	*Field assignment*
$\mid C\ x;$	*Local variable declaration*
$\mid x\texttt{=}e;$	*Variable assignment*
$\mid \texttt{return}\ e;$	*Return*
$\mid \texttt{\{}s_1 ... s_n\texttt{\}}$	*Block*

Middleweight Java is imperative and does not add any new control flow structure as compared to C. We thus map each AST element in Middleweight Java to an AST element in C. When compared to C, the three main additions of Middleweight Java are method invocation, object creation and constructor declarations. We map method invocations and object creations to function calls, and constructor declarations to function declarations. Class definitions are ignored as Coccinelle4J only allows matching and transformations of functions.

While we try to support the syntax of Java, we follow the goals of Coccinelle to decide the scope of our analysis on the source code [14]. One of Coccinelle's goals is to keep performance acceptable when used by developers on their development laptops. Thus, we always favor decisions keeping Coccinelle4J fast.

When the source code references an identifier, we do not attempt to resolve its type information that is not immediately obvious from the file getting parsed, such as the interfaces implemented by its class or its class hierarchy. We note that our choice contrasts with the design of some popular Java code-manipulation tools. For example, both Spoon [25] and Soot [27] provide an option for the user to pass the tool a classpath, allowing developers to specify where third-party library method and types can be found. Our choice is motivated by the statistics of widespread change in Table 1, where changes related to modifications in sub-typing or inheritance only comprise 6% of commits making widespread changes. When these transformations are required, a more appropriate tool such as Spoon can be used instead, as Coccinelle4J is complementary to it. We further discuss this decision and other implications of only allowing a limited form sub-typing and inheritance in Phase 3 (Section 3.3), the performance of Coccinelle4J in Section 4, and a brief comparison to Spoon in Section 4.8.

3.2 Phase 2: Control-flow with Exceptions and Java-specific Isomorphisms

Our second phase of the development of Coccinelle4J adds exceptions in the context of `try-catch`, to allow Coccinelle4J to parse and process a wider range of Java programs. To further improve the usability of Coccinelle4J, we add Java-specific isomorphisms, allowing some kinds of Java pattern code to match related syntactic variants.

3.2.1 Control-flow with exceptions

Middleweight Java does not allow code to throw exceptions, other than `ClassCastException` and `NullPointerException`, and does not provide syntax for `try` and `catch`. While this may be convenient for analysis, exceptions are used heavily in real-world Java programs. For example, about 25% of the Java classes in the source code of the five Java libraries previously discussed in Section 3 contain exception handling code. To allow Coccinelle4J to be more useful in practice, we add support for handling exceptions in the context of `try-catch`. Coccinelle4J does not deal with code throwing exceptions out of a method as it only performs intraprocedural analysis.

Given the code snippet in Listing 2, adapted from the code found in the NextCloud Android project,[10] we construct the control flow graph for the `try` statement as shown in Figure 2. As Coccinelle4J ignores the signatures of invoked methods from project dependencies and even within the same project, it does not know if any of the methods or constructors will

[10] `https://github.com/nextcloud/android`

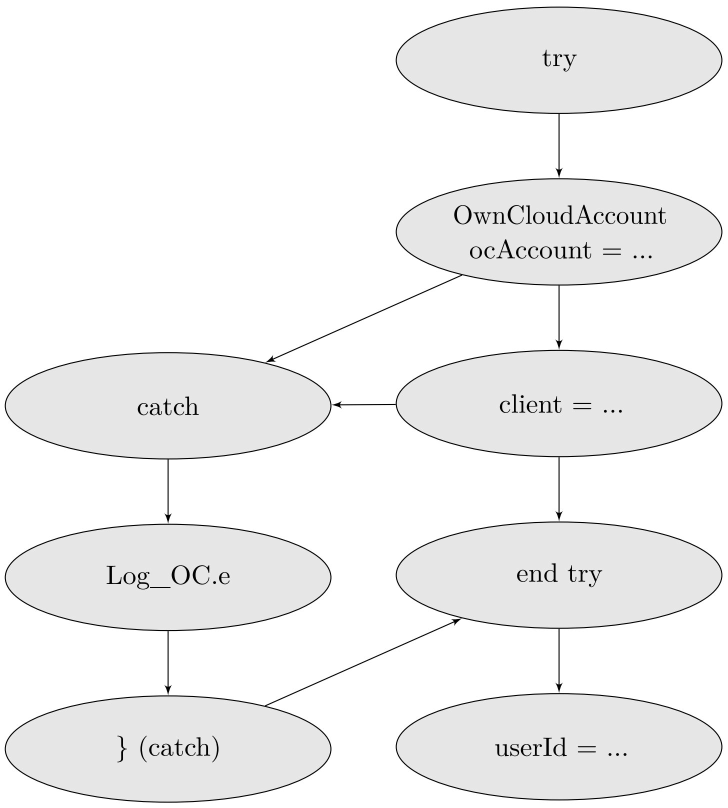

Figure 2 Control flow graph of try-catch

throw an exception. It thus over-approximates the set of paths that can be taken, adding edges between all the nodes in the `try` block and the start of the `catch` block.

Listing 2 Code snippet from the NextCloud Android project, modified for conciseness

```java
public RemoteTrashbinRepository(Context context) {
    AccountManager accountManager = AccountManager.get(context);
    Account account = AccountUtils.getCurrentOwnCloudAccount(
        context);
    try {
        OwnCloudAccount ocAccount = new OwnCloudAccount(account,
            context);
        client = OwnCloudClientManagerFactory.getDefaultSingleton()
            .getClientFor(ocAccount, context);
    } catch (Exception e) {
        Log_OC.e(TAG, e.getMessage());
    }
    userId = accountManager.getUserData(account,
        com.owncloud.android.lib.common.accounts.AccountUtils.
        Constants.KEY_USER_ID);
}
```

There are some ramifications of performing the control flow analysis in this way. For example, consider the following code snippet (Listing 3) and semantic patch (Listing 4), where `thisWontThrow` will never throw an `ExceptionA`. The control-flow graph created by Coccinelle4J has a path where `reachedHere` is reached. As the match succeeds, the Coccinelle adds the new invocation of `log.info`. However, in reality, the code in the `catch (ExceptionA e)` block is unreachable.

Listing 3 Hypothetical code where thisWontThrow() will not throw an exception

```
ObjectA A = new ObjectA();
try {
  A.thisWontThrow();
  reachedEnd = true;
} catch (ExceptionA e) {
  A.reachedHere();
}
```

Listing 4 Semantic patch adding a log message after A.reachedHere

```
@@
@@
A.thisWontThrow();
... when exists
A.reachedHere();
+ log.info("reached end of catch block");
```

In practice, we do not expect this to be common in real projects or be surprising to a user of Coccinelle4J. If `ExceptionA` is a checked exception, the Java compiler reports an error if the exception is never thrown in the `try` statement. For unchecked exceptions, as method signatures do not contain information about them, having the method signatures of the invocations in the `try` block will not help us prevent over-approximating the set of possible paths.

3.2.2 Isomorphisms

The fact that SmPL expresses patterns in terms of source code makes it easy to learn, but risks tying the specification to the coding style habits of a given developer. To address this issue, Coccinelle's engine handles *isomorphisms* during matching. Isomorphisms describe semantically equivalent code fragments that are used to expand SmPL rules, allowing syntactically different source code with the same semantics to be matched. For example, it allows `i++` (postfix increment) to be successfully matched against queries for `i=i+1`.

In Coccinelle4J, we disable Coccinelle isomorphism rules that are not relevant for Java (for example, pointer-related isomorphisms), and add Java-specific rules such as considering `A.equals(B)` to be semantically identical to `B.equals(A)`.

One particular difficulty we face in Java is that fully qualified names are considered by Coccinelle4J to be different from non-fully qualified ones even when referring to the same enumeration or method. This results in long patches specifying the same name multiple times using different qualifiers. To resolve this issue, we add a preprocessor to generate isomorphisms based on how names are qualified in a project. For example, when the rules in Listing 5 are enabled, users can specify "sqrt" in SmPL to match `Math.sqrt` or `java.lang.Math.sqrt` in the Java source code. This allows for more concise semantic patches. From our analysis of the 5 Java libraries previously discussed at the start of Section 3, less than 10% of method names and enumeration names were not unique when considering all of the projects together. Therefore, in many contexts, users do not need to distinguish between invocations of these different types of methods as there is often little risk in matching undesired function invocations of the same name.

Listing 5 Generated isomorphism rules, named java_lang_Math_sqrt and Math_sqrt, conflating qualified names and non-qualified names

```
Expression
@java_lang_Math_sqrt@
@@
sqrt => java.lang.Math.sqrt

Expression
@Math_sqrt@
@@
sqrt => Math.sqrt
```

Isomorphisms can be disabled on an individual basis within each rule by specifying `disable <isomorphism name>` beside the rule name or by passing a command line option to Coccinelle4J. The isomorphisms generated for matching fully qualified names are named "<package name>_<class name>_<identifier>" with the "." in the package name replaced by "_". As such, the rules in Listing 5 can be disabled by specifying `disable java_lang_Math_sqrt` and `disable Math_sqrt`. New isomorphisms can also be added by users.

3.3 Phase 3: Matching programs with sub-typing

Java projects often make use of sub-typing. Developers writing semantic patches may expect that a variable can be matched based on both its declared interface and the type specified in its constructor. Therefore, we permit a limited form of matching on variable types. Coccinelle4J annotates a variable using both its declared interface and its constructor. As a simple example, the metavariable `x`, declared with `ArrayList<Integer>` in the patch in Listing 6 can bind to the variable `numbers` in the statements in Listing 7. Coccinelle originally would only match if `x` was declared with `List<Integer>` in the semantic patch.

Listing 6 Semantic patch matching identifiers of type ArrayList<Integer>

```
@@
ArrayList<Integer> x;
identifier f;
@@
* f(x)
```

Listing 7 Statements with a List<Integer> variable

```
List<Integer> numbers = new ArrayList<Integer>();
doSomething(numbers);
```

Coccinelle4J's support for subtyping builds on Coccinelle's support for `typedef`. In C, `typedef` allows developers to introduce synonyms for types. For example, `typedef short s16;` introduces a new type alias `s16` for the type `short`. During program matching, developers may expect that writing `short` in SmPL matches expressions of type `s16` in the source code.

Coccinelle already resolves type information related to `typedef`s during program matching. When parsing C programs, Coccinelle tracks type information for each `typedef` and annotates each program element in the C source code with the types that it can be matched on. Whenever an element declared as a typedef is reached, Coccinelle tries to resolve the type information from the relevant `typedef` declaration. In Coccinelle4J, we reuse this mechanism to deal with sub-typing in Java, maintaining a list of types that each program element can be matched on.

One limitation is that we only match interfaces to object instances of their sub-types if the variables were explicitly declared with their interface. For example, `numbers` in Listing 7 implements multiple interfaces including `List<Integer>` and `Iterable<Integer>`. In Coccinelle4J, we only match `numbers` on the explicitly declared types `List<Integer>` and `ArrayList<Integer>`. Information such as the inheritance path of the identifier's class is also ignored; `numbers` inherits `AbstractList<Integer>`, `AbstractCollection<Integer>`, and `Object`, but Coccinelle4J will not match them. As a result of this decision, in Listing 8, patterns specifying `Deque<UserInfoDetailsItem>` will not match any statement in the Listing even though `LinkedLists` satisfy the `Deque` interface, since it is not declared as a `Deque` here.

Listing 8 Code snippet modified from NextCloud Android. This code snippet matches `List<UserInfoDetailsItem>` or `LinkedList<UserInfoDetailsItem>`, but not `Deque< UserInfoDetailsItem>`

```
List<UserInfoDetailsItem> result = new LinkedList<>();
addToListIfNeeded(result, R.drawable.ic_phone, userInfo.getPhone(),
                    R.string.user_info_phone);
return result;
```

In practice, allowing a semantic patch specifying an interface/parent class to match on all uses of sub-types/child classes may result in many unwanted results. Classes may implement multiple interfaces, and when given a particular context, it is improbable that every implemented interface is relevant. In many situations where matching an interface or parent class is intended, specifying the interface or its methods will suffice. In most cases, matching on explicit declarations is sufficient, since it is idiomatic in Java to "Refer to objects by their interfaces" [2]. For example, `numbers` is declared with its interface `List<Integer>` in Listing 7 and `result` is declared with `List<UserInfoDetailsItem>` in Listing 8.

We verified this idiom empirically. In the Java libraries we studied, 75% of method parameters and local variable declarations were declared with an interface, whenever such an interface (excluding low-level interfaces from `java.lang` and `java.io`) existed. In cases where variables are not declared with their interfaces and we wish to detect them, specifying the methods on the interface can be attempted next. For example, the `Cloneable` interface is implemented by numerous classes. If the user intends to find parts of programs where objects are cloned, it is sufficient to specify the invocation of the `clone()` method in the semantic patch.

Another reason for this decision is that resolving the type of every identifier will incur a performance penalty, and we would like to avoid the cost of trying to resolve project dependencies or having to fetch type information from third-party libraries. Coccinelle4J is designed to be used by developers for rapid prototyping, one rule at a time, so that users can receive feedback from the tool quickly. Thus, one of our goals is to keep Coccinelle4J lightweight.

4 Case Study: Migrating APIs

Android applications may have many dependencies on third-party libraries or frameworks, and keeping up with changes in those dependencies may introduce a significant maintenance burden. Due to security or performance issues, libraries or frameworks deprecate API methods and these methods are eventually removed [16]. It is therefore important to upgrade the uses of deprecated methods to their more modern alternatives. McDonnell et al. studied the impact of API evolution of Android on developer adoption [17], and they found that it takes time for developers to catch up with new API versions. They also observed that API updates are more probable to lead to defects than other changes. We note that this shows the importance of providing developers with a faster and more reliable way of updating API usage.

We show examples of deprecated Android API methods with typical migration pathways, and show possible semantic patches for these migrations. Using these examples, we show that typical migrations can be written concisely in SmPL. We order these examples in approximately increasing order of complexity. In Section 4.1, 4.2, and 4.3, we show basic features of Coccinelle4J to replace method invocations, arguments to them, and their method receivers. In Section 4.4, we describe a case requiring a change in logic due to a different return type of the replacement API, and in Section 4.5, we distinguish between the different ways the deprecated API can be used based on its context. In Section 4.6, we replace the parameters of a method overriding a deprecated method from its parent class. Finally, we discuss an example where we encounter some false positives during program matching due to the limitations described earlier.

As some method invocations may have numerous call sites across a project, migration of these API methods is a form of widespread change. We show that the use of semantic patches may reduce developer effort. This also serves as a short guide on specifying program transformations in SmPL, and we show how context information may be useful. While performing the migrations, there are two questions that we wish to answer:

- Is SmPL expressive enough to perform widespread changes in Java projects?
- How much development effort can a developer potentially save by the use of semantic patches?

4.1 Removing sticky broadcasts

We use SmPL to demonstrate two related trivial migrations of replacing invocations of a deprecated method while keeping the same arguments, and removing the invocations of another method.

The use of sticky broadcasts was discouraged in the release of Android API level 21 due to problems such as the lack of security, therefore the methods `sendStickyBroadcast(Intent)` and `removeStickyBroadcast(Intent)` were deprecated. The official API reference[11] recommends `sendBroadcast(Intent)` to replace `sendStickyBroadcast(Intent)`, while usage of `removeStickyBroadcast(Intent)` should be removed. It was suggested that the developer should also provide some mechanism for other applications to fetch the data in the sticky broadcasts. Providing this latter mechanism is highly dependent on the Android application, and requires knowledge of what the broadcast receivers expect of the broadcast. However, replacing and removing these method calls can be expressed in SmPL.

Listings 9 and 10 illustrate the use of these API methods in the NextCloud Android project, while Listing 11 shows a semantic patch that handles this migration. The call site of `removeStickyBroadcast` is an example motivating that migration of function invocations on exceptional control flow may be required, as `removeStickyBroadcast` may be included in a `finally` block to ensure that it always executes. In this patch, we use the '|' operator, *i.e.*, the *sequential disjunction* operator of SmPL, to remove invocations of `removeStickyBroadcast`. The '|' operator allows matching and transformation using any of the terms delimited by '|' within the parenthesis. Earlier matches take precedence over later ones, for a given node

[11] `https://developer.android.com/reference/android/content/Context.html#sendStickyBroadcast(android.content.Intent)`

in the control-flow graph. In this case, the rule matches on either `sendStickyBroadcast` or `removeStickyBroadcast`. This allows for the succinct expression of multiple matching code that have similar contexts.

Listing 9 Example call site of sendStickyBroadcast in the NextCloud Android project

```
private void sendBroadcastUploadsAdded() {
    Intent start = new Intent(getUploadsAddedMessage());
    // nothing else needed right now
    start.setPackage(getPackageName());
    sendStickyBroadcast(start);
}
```

Listing 10 Example call site of removeStickyBroadcast in the NextCloud Android project

```
try {
    ... // omitted for brevity
    removeStickyBroadcast(intent);
    Log_OC.d(TAG, "Setting progress visibility to " + mSyncInProgress);
} catch (RuntimeException e) {
    // comments omitted for brevity
    removeStickyBroadcast(intent);
    DataHolderUtil.getInstance().delete(
        intent.getStringExtra(FileSyncAdapter.EXTRA_RESULT));
}
```

Listing 11 Semantic patch for migrating from sendStickyBroadcast

```
@@
Intent intent;
@@
(
- sendStickyBroadcast(intent);
+ sendBroadcast(intent);
|
- removeStickyBroadcast(intent);
)
```

4.2 setTextSize → setTextZoom

Li et al. have inferred a collection of mappings from deprecated Android API methods to their replacement methods [16]. A typical migration in this collection requires modifying both method names and method arguments, either by adding or removing arguments, or by changing the arguments' type. One such transformation was required when `WebSettings.setTextSize(WebSettings.TextSize)` was deprecated and replaced by `WebSettings.setTextZoom(int)`. This involves transforming both the method name and the method arguments. Listing 12 shows part of the patch on the Lucid-Browser project[12] and Listing 13 then shows a semantic patch producing it. The semantic patch modifies both the method's name and its arguments in a single rule.

Listing 12 Part of a patch replacing invocations of setTextSize in the Lucid-Browser project

```
if (Properties.webpageProp.fontSize==0)
```

[12] `https://github.com/powerpoint45/Lucid-Browser`

```
-   this.getSettings().setTextSize(WebSettings.TextSize.SMALLEST);
+   this.getSettings().setTextZoom(50);
if (Properties.webpageProp.fontSize==1)
-   this.getSettings().setTextSize(WebSettings.TextSize.SMALLER);
+   this.getSettings().setTextZoom(75);
if (Properties.webpageProp.fontSize==2)
-   this.getSettings().setTextSize(WebSettings.TextSize.NORMAL);
+   this.getSettings().setTextZoom(100);
if (Properties.webpageProp.fontSize==3)
-   this.getSettings().setTextSize(WebSettings.TextSize.LARGER);
+   this.getSettings().setTextZoom(150);
if (Properties.webpageProp.fontSize==4)
-   this.getSettings().setTextSize(WebSettings.TextSize.LARGEST);
+   this.getSettings().setTextZoom(200);
```

Listing 13 Semantic patch replacing usage of setTextSize

```
@@
expression E;
@@
(
- E.setTextSize(LARGEST);
+ E.setTextZoom(200);
|
- E.setTextSize(LARGER);
+ E.setTextZoom(150);
|
- E.setTextSize(NORMAL);
+ E.setTextZoom(100);
|
- E.setTextSize(SMALLER);
+ E.setTextZoom(75);
|
- E.setTextSize(SMALLEST);
+ E.setTextZoom(50);
)
```

This example illustrates the usefulness of the isomorphisms conflating fully qualified and non-fully qualified class names. While in most projects, `WebSettings.TextSize.LARGEST` is often qualified with `WebSettings.TextSize`, some projects, such as K-9 Mail,[13] use the shorter `TextSize.LARGEST`. Listings 14 and 15 show the isomorphism rules generated for `LARGEST`.

Listing 14 Isomorphism rule for expanding LARGEST for Lucid Project

```
Expression
@ WebSettings_TextSize_LARGEST @
@@
LARGEST => WebSettings.TextSize.LARGEST
```

Listing 15 Isomorphism rule for expanding LARGEST for K-9 Mail

```
Expression
@ TextSize_LARGEST @
@@
LARGEST => TextSize.LARGEST
```

[13] `https://github.com/k9mail/k-9/commit/f8695f9a61c8a411a09ccee8c8bf739149f0f17e`

4.3 Resources.getColor → ContextCompat.getColor

Another migration pathway is changing the method receiver. We use the deprecation of `Resources.getColor(int)` as an example. One approach to replace this deprecated method is to use the static method `ContextCompat.getColor(Context, int)`, which was specifically introduced to help migrate from the deprecated method. A commit making such a change can be found in the Kickstarter Android project.[14] A patch, similar to that commit, that can be applied on Kickstarter Android is in Listing 16 and the semantic patch producing this patch is in Listing 17.

Listing 16 Part of a patch replacing getColor from Kickstarter Android

```
return new NotificationCompat.Builder(context)
      .setSmallIcon(R.drawable.ic_kickstarter_k)
-     .setColor(context.getResources().getColor(R.color.green))
+     .setColor(ContextCompat.getColor(context, R.color.green))
      .setContentText(text)
      .setContentTitle(title)
      .setStyle(new NotificationCompat.BigTextStyle().bigText(text))
      .setAutoCancel(true);
```

Listing 17 Semantic patch replacing uses of getColor

```
@@
Context ctx;
expression E;
@@
- ctx.getResources().getColor(E)
+ ContextCompat.getColor(ctx, E)
```

4.4 AudioManager.shouldVibrate(int) → AudioManager.getRingerMode()

In some cases, migration requires more changes than just replacing the deprecated method with another and requires additional logic to be added by the developer. For example, the migration pathway for `AudioManager.shouldVibrate` was to replace it with a comparison of the ringer mode retrieved from `AudioManager.getRingerMode`. This deprecation required an application to maintain its own policy for allowing vibration of the phone based on the phone's current ringer mode. Support for both older and newer Android versions may be kept by using the deprecated method only on earlier Android versions, while using the replacement method on later Android versions. An example showing the result of the required transformation can be found in the Signal Android application[15] shown in Listing 18. Listing 19 shows a semantic patch that adds a new function that dispatches a call to the right method after checking the Android version, and replaces the deprecated method invocation with this method. A default policy of allowing vibration on non-silent modes is assumed in this patch, but developers can modify the semantic patch to customize the vibrate policy for their own applications appropriately.

[14] https://github.com/kickstarter/android-oss/commit/053b0a32731bd9a4e9dd42c297565f87145a964b
[15] https://github.com/signalapp/Signal-Android/blob/f9adb4e4554a44fd65b77320e34bf4bccf7924ce/src/org/thoughtcrime/securesms/webrtc/audio/IncomingRinger.java

Listing 18 Example code invoking the deprecated method from Signal Android

```java
private boolean shouldVibrate(Context context, MediaPlayer player,
        int ringerMode, boolean vibrate) {
    if (player == null) {
      return true;
    }

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
      return shouldVibrateNew(context, ringerMode, vibrate);
    } else {
      return shouldVibrateOld(context, vibrate);
    }
  }

  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  private boolean shouldVibrateNew(Context context,
          int ringerMode, boolean vibrate) {
    Vibrator vibrator = (Vibrator) context.getSystemService(
        Context.VIBRATOR_SERVICE);

    if (vibrator == null || !vibrator.hasVibrator()) {
      return false;
    }

    if (vibrate) {
      return ringerMode != AudioManager.RINGER_MODE_SILENT;
    } else {
      return ringerMode == AudioManager.RINGER_MODE_VIBRATE;
    }
  }

  private boolean shouldVibrateOld(Context context, boolean vibrate) {
    AudioManager audioManager = ServiceUtil.getAudioManager(context);
    return vibrate &&
        audioManager.shouldVibrate(AudioManager.VIBRATE_TYPE_RINGER);
  }
```

Listing 19 Semantic Patch replacing uses of shouldVibrate

```
@@
identifier am, f, ctx;
expression vibrate_type;
@@
+ boolean shouldVibrate(AudioManager am, Context ctx, int vibrateType) {
+     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
+         Vibrator vibrator = (Vibrator) ctx. getSystemService(
+             Context.VIBRATOR_SERVICE );
+         if (vibrator == null || !vibrator.hasVibrator()) {
+                 return false;
+         }
+         return am.getRingerMode() != AudioManager.RINGER_MODE_SILENT;
+     } else {
+         return audioManager.shouldVibrate(vibrateType);
+     }
+ }
f(..., Context ctx, ...) {
...
- am.shouldVibrate(vibrate_type)
+ shouldVibrate(am, ctx, vibrate_type)
...
}
```

4.5 Display.getHeight() and Display.getWidth() → Display.getSize(Point)

Deprecated methods can be used in different ways, and migration requires consideration of how they were used. For example, in the release of Android API Level 15, `Display.getHeight` and `Display.getWidth` were deprecated in favor of constructing a `Point` object, initializing it using `Display.getSize(Point)`, before obtaining the height and width using `Point.y` and `Point.x`. Listings 20 and 21 show two examples of code (from the Materialistic for Hacker News application[16] and the Glide library[17]) managing the deprecation. Both examples check for the currently installed Android version, and invoke the deprecated method only on earlier versions of Android where the method has not been deprecated.

These two examples show the need to distinguish between code requiring a `Point` object instance, and code requiring just the height of a display. In Listing 21, a `Point` object is already constructed and only the way it is initialized requires modification. In Listing 20, however, only the height of the display is required. Listing 22 shows a simplified semantic patch that consists of two rules to fix the two variants of this deprecation. The tokens `<...` and `...>` form a variant of "`...`" indicating that matching `display.getHeight()` within the path is optional and can occur multiple times. In this case, we use `<... ...>` to specify that we wish to transform all of its occurrences. The use of `... when != Point(...)` in rule2 omits matches where a `Point` object has already been created in the control flow context, in order to distinguish between the two cases. Listing 23 shows one example patch produced by Coccinelle4J that is semantically similar to a commit fixing the deprecation in the Android framework itself.[18]

Listing 20 Example from Materialistic of correct usage of the deprecated method, getHeight, after checking for the device's Android version

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {
    displayDimens = new Point();
    display.getSize(displayDimens);
} else {
    displayDimens = new Point(display.getWidth(), display.getHeight());
}
... // omitted for brevity
```

Listing 21 Example from Glide of correct usage of the deprecated method, getHeight, after checking for the device's Android version

```
... // omitted for brevity
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {
    Point point = new Point();
    display.getSize(point);
    return point.y;
} else {
    return display.getHeight();
}
```

Listing 22 Simplified version of a semantic patch to migrate invocations of getWidth and getHeight. The use of != Point(...) prevents rule2 from erroneously matching code transformed for the first case.

[16] https://github.com/hidroh/materialistic/commit/baaf9eeb28f09b8d0d875107687a595a2f377f79
[17] https://github.com/bumptech/glide/commit/827fc08222eb61595ab0d5fdebeea0033e9e8382
[18] https://github.com/aosp-mirror/platform_frameworks_base/commit/ac8dea12c17aa047e03a358110aeb60401d36aa2#diff-40287e2f0fc8c327021400d034442324

```
@rule1@
Display display;
identifier p;
type T;
@@
(
- p = new Point(display.getWidth(),
-               display.getHeight());
+ p = new Point();
+ display.getSize(p);
|
- T p = new Point(display.getWidth(),
-                 display.getHeight());
+ T p = new Point();
+ display.getSize(p);
)
// '<...' indicates that all occurrences should be replaced
<...
(
- display.getHeight()
+ p.y
|
- display.getWidth()
+ p.x
)
...>

@rule2@
identifier display, f;
expression E;
@@
f(...) {
...
Display display = E;
+ Point p = new Point();
+ display.getSize(p);
// 'when' omits matches where a Point object has already been created
<... when != Point(...)
(
- display.getHeight()
+ p.y
|
- display.getWidth()
+ p.x
)
...>
}
```

Listing 23 Example patch created by Coccinelle4J using the semantic patch above, semantically equivalent to a commit fixing this deprecation in the Android framework

```
  public static void dragQuarterScreenDown(
            InstrumentationTestCase test, Activity activity) {
   Display display = activity.getWindowManager().getDefaultDisplay();
-     int screenHeight = display.getHeight();
-     int screenWidth = display.getWidth();
+     Point p = new Point();
+     display.getSize(p);
+     int screenHeight = p.y;
+     int screenWidth = p.x;
```

4.6 WebChromeClient.onConsoleMessage(String, int, String) → WebChromeClient.onConsoleMessage(ConsoleMessage)

We show how method signatures of classes extending other classes with deprecated methods can be transformed using SmPL. Developers can override `onConsoleMessage(String, int, String)` of `WebChromeClient`, which was deprecated in favour of an overloaded method with a different set of parameters `onConsoleMessage(ConsoleMessage)`. Listing 24 shows example code from the MGit project[19] overriding this method, and Listing 25 shows a semantic patch changing the parameters of `onConsoleMessage(String, int, String)`. The use of `<...` and `...>` specifies that all occurrences of `p1`, `p2`, and `p3` should be transformed.

Listing 24 An example replacement site of onConsoleMessage from MGit

```
mFileContent.setWebChromeClient(new WebChromeClient() {
    @Override
    public void onConsoleMessage(String message, int lineNumber,
                                 String sourceID) {
        Log.d("MyApplication", message + " -- From line " + lineNumber
                + " of " + sourceID);
    }
... // other overridden methods
})
```

Listing 25 Semantic patch updating method signatures of onConsoleMessage

```
@@
identifier p1, p2, p3;
@@
- onConsoleMessage(String p1, int p2, String p3) {
+ onConsoleMessage(ConsoleMessage cs) {
<...
(
- p1
+ cs.message()
|
- p2
+ cs.lineNumber()
|
- p3
+ cs.sourceId()
)
...>
}
```

4.7 Resources.getDrawable(int) → Resources.getDrawable(int, Theme)

In Phase 3 of developing Coccinelle4J, we made some trade-offs that introduced the limitations discussed in Section 3. While we do not run into problems writing the majority of the patches, some patches would have benefited from Coccinelle4J having the ability to resolve types, one of the features deliberately omitted to keep Coccinelle4J fast. In the release of Android API level 22, `Resources.getDrawable(int)` was deprecated in favour of `getDrawable(int, Theme)`, the same method overloaded with an additional `Theme` parameter. After inspecting calls to `getDrawable` in a project, a developer may write a semantic patch to find these calls and transform them using the simple patch in Listing 26.

[19] `https://github.com/maks/MGit`

Listing 26 Semantic patch updating invocations to getDrawable

```
@@
expression E;
expression p;
@@
- E.getDrawable(p)
+ E.getDrawable(p, getContext().getTheme())
```

While all the locations that should be transformed are correctly transformed, this also results in several false positives, as illustrated by the patch in Listing 27, based on code from the NextCloud Android project. As the semantic patch does not assert that the metavariable `E` is of type `Resources`, Coccinelle4J updates a `getDrawable` invocation on a `TypedArray` instead of `Resources`. This results in code that does not compile as `TypedArray` does not have a method with a `getDrawable(int, Theme)` signature.

Listing 27 Example patch created by Coccinelle4J for the above semantic patch to replace getDrawable

```
     public SimpleListItemDividerDecoration(Context context) {
         super(context, DividerItemDecoration.VERTICAL);
         final TypedArray styledAttributes =
            context.obtainStyledAttributes(ATTRS);
-        divider = styledAttributes.getDrawable(0);
+        divider = styledAttributes.getDrawable(0,
+            getContext().getTheme());
         leftPadding = Math.round(72 *
            (context.getResources().getDisplayMetrics().xdpi /
            DisplayMetrics.DENSITY_DEFAULT));
         styledAttributes.recycle();
     }
```

On the other hand, if one does specify the need for a `Resource`, as illustrated in Listing 28, then Coccinelle4J misses locations where a `Resource` variable is not explicitly declared. For example, a `Resource` may be produced from an invocation as part of a larger expression, as in `getResources().getDrawable(R.drawable.ic_activity_light_grey)`. The metavariable `Resources R` will not match the invocation of `getResources`. In cases like this, developers may be surprised that there is no way to write a rule in SmPL with a metavariable binding to the invocation of `getResources` based on its return type.

Listing 28 Semantic patch updating invocations to getDrawable

```
@@
Resources R;
expression p;
@@
- R.getDrawable(p)
+ R.getDrawable(p, getContext().getTheme())
```

4.8 Evaluation

Using the case of performing changes to multiple locations, we evaluate Coccinelle4J based on how much it helped development efficiency, the ease of specifying patches, and the speed at which it applied patches to projects. We also report limitations we experienced with Coccinelle4J.

We use development efficiency metrics that were previously used to evaluate Coccinelle for backporting device drivers [26]. Table 2 shows the ratio of the number of source-code insertions and deletions that Coccinelle4J generates, the number of lines in the semantic

Table 2 Summary of the patches generated by Coccinelle4J in our case study

Patch	Semantic patch size	Lines generated	Ratio	Files changed
sendStickyBroadcast	9	19	2.11	5
setTextSize	19	10	0.53	1
getColor	6	16	2.67	4
shouldVibrate	35	17	0.49	1
getHeight	44	3	0.07	1
onConsoleMessage	18	14	0.78	2

patch, excluding whitespace, and the number of files changed by Coccinelle4J. Comparing these quantities indicates the amount of savings from the use of semantic patches. We use the patches updating the uses of `sendStickyBroadcast` on the NextCloud Android project (Section 4.1),[20] `setTextSize` on the Lucid-Browser (Section 4.2), `getColor` on the Kickstarter Android project (Section 4.3), `shouldVibrate` on Signal (Section 4.4), `getHeight` on Glide (Section 4.5), and `onConsoleMessage` on the MGit project (Section 4.6). In total, these projects contain 311755 lines of code and 2030 source files.

The development effort saved of using Coccinelle4J increases in proportion with the number of replacement sites. For changes that have to be made at multiple locations, Coccinelle4J will save more effort. In our examples, as there were only a few uses of `shouldVibrate`, `getHeight`, and `onConsoleMessage` in MGit, Signal, and Glide, the number of lines of code in the semantic patch is greater than the number of lines of code of the actual change. However, the amount of code changed is an incomplete measure of the development effort saved, as Coccinelle4J also helps in locating the files requiring modification, as well as in identifying the correct locations in each file. Furthermore, as these are methods of the Android API, they will be commonly used by many projects, and each patch will have numerous matches when considering all Android repositories.

In all our examples, the semantic patches to update the call sites of the deprecated methods are concise. The semantic patches are declarative and describe what code will be changed after their application to the source code.

We briefly compare the expressiveness of SmPL to Spoon [25]. With Spoon, there are currently no abstractions over control-flow constraints. Transformations in Spoon are less declarative compared to SmPL. Using the example of `onConsoleMessage` (Listing 25) discussed earlier in Section 4.6, where we change its parameters, Listing 29 shows part of a processor written in Spoon. As compared to the semantic patch, the Spoon processor is imperative and requires reasoning about Spoon's meta-model, whereas SmPL takes a WYSIWYG approach.

Listing 29 Replacing parameters of onConsoleMessage using Spoon

```
    // ...
    // Code for selecting the onConsoleMessage method omited
    public void process(CtMethod method) {
        List<CtParameter> params = method.getParameters();
        while (!params.isEmpty()) {
            CtParameter param = params.get(0);
            param.delete();
            params.remove(param);
```

[20] `https://github.com/nextcloud/android`

```
        }
        CtParameter<ConsoleMessage> newParam = getFactory().Core()
            .createParameter();
        newParam.setSimpleName("consoleMessage");
        newParam.setType(getFactory().Type()
            .createReference(ConsoleMessage.class));
        method.addParameter(newParam);
        // ...
    }
    // ...
```

Listing 30 Replacing parameters of onConsoleMessage using SmPL

```
@@
identifier p1, p2, p3;
@@
- onConsoleMessage(String p1, int p2, String p3) {
+ onConsoleMessage(ConsoleMessage consoleMessage) {
...
}
```

We further elaborate on other convenience features Coccinelle4J provides. While Spoon has templates for transformation, developers have to specify template parameters and write logic in Spoon processors to extract or construct the corresponding arguments. Using the example discussed in Section 4.2 about replacing `setTextSize` with `setTextZoom`, we define a Spoon template in Listing 31. This template has two parameters, an expression returning a `WebSettings` that `setTextSize` is invoked on, and a literal that is passed as an argument to `setTextSize`. A Spoon processor (Listing 32) has to select these meta-model elements from the matching statement, and pass them as arguments to the template to generate a new statement, before replacing the original statement invoking `setTextSize`. The corresponding patch in SmPL was given previously in Section 4.2. In contrast to Spoon, a user of Coccinelle4J does not need to think about these low-level details, as Coccinelle4J binds metavariables to subterms while matching code and automatically applies them during transformation.

Listing 31 Spoon template for producing statements using setTextZoom

```
public class ReplaceTemplate extends StatementTemplate {
    public ReplaceTemplate(CtExpression<WebSettings> _settings,
                CtLiteral<Integer> _value) {
        this._settings = _settings;
        this._value = _value;
    }

    @Parameter
    CtExpression<WebSettings> _settings;

    @Parameter
    CtLiteral<Integer> _value;

    @Override
    public void statement() {
        _settings.S().setTextZoom(_value.S());
    }
}
```

Listing 32 Spoon processor using the Spoon template replacing invocations of setTextSize

```
public class SetTextzoomProcessor
        extends AbstractProcessor<CtStatement> {
    ... // other details omitted for brevity
    @Override
    public void process(CtStatement stmt) {
        ... // omitted code to select the expressions:
        // 1. oldValue refers to the argument of setTextSize
        // 2. settingsExpression refers to the expression
        // returning the WebSettings that setTextSize is invoked on.
        if (oldValue.getVariable().getSimpleName().equals("SMALLEST")
            ) {
            value.setValue(50);
        } else if (oldValue.getVariable().getSimpleName()
                    .equals("SMALLER")) {
            value.setValue(75);
        } else if (...) {
            ...
        }
        ... // more code to set a correct value for the template

        ReplaceTemplate template = new ReplaceTemplate(
            settingsExpression, value);
        CtStatement newstmt = template.apply(stmt.getParent(
            new TypeFilter<>(CtType.class)));
        stmt.replace(newstmt);
   }

}
```

We also briefly compare Coccinelle4J to Refaster [31], a tool to refactor code by writing templates of code before and after its transformation. While both SmPL and Refaster templates allow developers to declaratively refine transformation rules, the design of Coccinelle4J allows reasoning on control-flow paths and fundamentally differs from Refaster. The practical advantages are:

- Constraints can be specified on control-flow paths and matching is supported within arbitrary control-flow paths (including around loops), see Listing 1,
- Metavariables can have different values in different control-flow paths, while being forced to have consistent values within a control-flow path,
- Coccinelle4J interleaves addition or removal of code in the same rule. This makes it easy to position code changes in a long series of code elements, see Listing 22.

It may not be easy to add these advantages, which are inherent in Coccinelle4J, to Refaster. Additionally, Refaster lacks several features of Coccinelle4J, such as:

- inheritance of metavariables between rules,
- interaction with scripting languages (OCaml or Python).

Performance-wise, Coccinelle4J runs quickly. Our experiments were performed on a 2017 Macbook Pro with 2.3 GHz Intel Core i5, 8 GB 2133 MHz LPDDR3, likely similar to an average developer's working laptop. We report the time required for Coccinelle4J to perform each transformation. We also report the time required to build the project, but exclude the time for downloading the project dependencies in the first build. We consider this time to be

Table 3 Time to apply semantic patch compared to time to build the project (rounded to the nearest second)

Patch	Project	Time to apply patch	Time to build project
sendStickyBroadcast	NextCloud	3s	46s
setTextSize	Lucid-Browser	1s	21s
getColor	Kickstarter	1s	1 min 6s
shouldVibrate	Signal	1s	1 min 54s
getHeight	Glide	0s	33s
onConsoleMessage	MGit	0s	57s

an upper bound on the time that project developers will wait for a tool to complete running. The results are given in Table 3.

In each case, the time to apply a semantic patch on an entire project is nearly negligible, even when there are multiple transformation sites, and it is only a small fraction of the time required to build the project. As such, we conclude that other than the time required to write a semantic patch, usage of the tool will not affect a developer's time negatively.

A limitation of Coccinelle4J is that users may write patches that gives false negatives while matching programs. When writing a semantic patch, it is often the case that one starts with a simple semantic patch that misses more complex cases. We believe that this limitation is circumvented as semantic patches are concise and developers can quickly refine a semantic patch after inspecting the output produced by Coccinelle4J. The time to apply the updated semantic patch to a project is also negligible. This short feedback loop allows for a fast iterative development process and facilitates exploratory programming, much like working with a read-eval-print loop (REPL). We find this aspect of development lacking in other program transformation tools.

On the other hand, Coccinelle4J inherits some limitations of Coccinelle. One painful aspect of working with Coccinelle4J is it does not always report errors when parsing SMPL in a user-friendly way. This limitation is inherent in yacc-like parsers. While Coccinelle reports the position and identifier in the semantic patch where the error was detected, it may not be immediately clear to a new user of Coccinelle what the error is or how to correct it. Editing semantic patches may be difficult due to the current lack of support in tools familiar to Java developers. For example, while Coccinelle provides some support for Vim and Emacs, there is lack of support for other popular text editors or IDEs that Java developers may be more familiar with. Without basic support that developers may be accustomed to, such as syntax highlighting or code completion, developers may make mistakes that are hard to notice. However, this limitation is circumvented by the ease of adding support for SmPL in text editors, due to the reuse of Java syntax as the code matching language. Providing such support may be a next step to improve the ergonomics of writing SmPL.

Since Coccinelle4J does not guarantee correctness, there is the danger that it may generate semantically incorrect patches (false positives). We believe that this risk is mitigated by two factors. Firstly, most incorrect patches will be caught by the Java compiler. As Coccinelle4J is targeted at changing code in multiple locations, we expect most patches produced by Coccinelle4J to fall into categories of widespread changes, which researchers have categorized for Java projects [30]. Code changes in most of these categories will result in compiler errors if mistakes are made. The iterative process of refining patches also allows mistakes to be detected and identified quickly. Developers can fix incorrect changes manually or revert them with revision control tools. Secondly, in large Java projects where widespread changes are most relevant, there are strict code review processes where errors will be caught by more experienced project maintainers. As such, despite not having the guarantee of producing correct patches, the use of Coccinelle4J will not negatively impact the software development process.

5 Related Work

5.1 Program matching and transformation for Java

There are several existing tools for Java program transformation. Refaster [31] and Spoon [25], already considered in the previous section, are designed to be easy for developers to use. Like Coccinelle4J, they work on the AST and match elements independent of white-space and formatting. However, unlike Coccinelle4J, neither Spoon nor Refaster takes control flow into account when matching code and does not allow for the specification of constraints over control-flow paths. More generally, there is no equivalent to the "..." abstraction in SmPL that abstracts over an arbitrary sequence of code. On the other hand, Spoon allows rules to reason about more precise type information. As such, our work on Coccinelle4J is complementary to the work on Spoon.

Stratego [28], Rascal [10], and TXL [6] are other tools for program transformation that can work on Java. However, these tools requires developers to invest time to learn syntax and formalisms that do not resemble Java.

Soot [27] is a framework for Java program optimization and also offers program analysis tools. Although it provides tools that can compute control flow graphs, it does not provide any program transformation tool that acts on these results.

Another class of Java transformation tools operate on bytecode, unlike Coccinelle4J which works on Java source code. JMangler [11], ASM [4] and Javassist[5] are examples of tools providing APIs for bytecode manipulation, that can be used for tasks like creating new class definitions derived from other classes.

5.2 Migration of APIs

There are several approaches to automated API usage updates. For example, Henkel and Diwan [8] presented a tool to capture API refactoring actions when a developer updates the API usage. LibSync [21] recommends potential edit operations for adapting code based on clients that have already migrated from an API. Semdiff [7] recommends calls to a framework based on how the framework adapted to its own changes. HiMa [18] performs pairwise comparisons in the evolutionary history of a project to construct framework-evolution rules.

Several works also propose program transformation languages to describe rules for mapping APIs calls to alternative APIs. Nita and Notkin describe Twinning [22], a rule-based language that allows developers to specify mappings of blocks of API invocations to sequences of alternative API invocations. SWIN [15] extends Twinning, including adding type safety of transformations. The type safety of SWIN is proved on Featherweight Java [9], a minimal core calculus similar to Middleweight Java.

The approaches above do not allow for context-sensitive many-to-many mappings. Many-to-many mappings refer to the transformation of a sequence of statements using an old API to a new sequence of statements using a replacement API. Wang et al. [29] highlight the difficulty of API migrations when these mappings are required. They show the need to account for control-flow as the statements requiring transformation may take multiple forms.

They propose *guided-normalisation* and a language *PATL* for transforming Java programs between different APIs. The semantics of PATL are formalised on Middleweight Java.

Their work differs from ours as their focus is transforming programs between two APIs, a single task where program transformation is useful, while our work targets the more general task of program transformation itself. We use migrations of deprecated APIs as an example only to demonstrate the utility of our program transformation tool. While transforming programs between APIs often focuses on short sequences of function invocations, Coccinelle4J can express constraints and transformations of the code over a longer range, including involving information collected from multiple field and method declarations. There is also no equivalent of the "..." operator in PATL.

An experience report about the use of automated API migration techniques suggests that the difficulty of API migrations lies in the change of API parameter types, rather than selecting an alternative API [13]. This is motivation for our work, as we support the transformation of code and arguments to methods based on the code context. Semantic patches allow argument values to be determined based on code context, including specifying constraints over paths, the use of multiple rules, and information collected from multiple fields and method declarations.

6 Conclusion and Future Work

While SmPL has been shown to be useful for program transformation on C code, its use has not been explored in the context of Java projects. Semantic patches have the benefit of being declarative, are relatively easy to specify, and are unique in how they allow the expression of control-flow patterns. To introduce a tool with these features into the Java ecosystem, we have developed Coccinelle4J, a prototype extending Coccinelle, to support some Java language features. We document our implementation and the design decisions made. Finally, we look at several cases of updating call sites of deprecated Android API in six projects to show the utility of Coccinelle4J. Based on this case study, we discuss its suitability for use in Java projects.

As future work, we hope to evaluate the use of semantic patches and Coccinelle4J for other uses, such as to fix common bugs. From our preliminary work on migrating deprecated Android API, a further extension may be to create a public dataset of reliable semantic patches that developers can apply to their Android projects.

References

1 Gavin M Bierman, MJ Parkinson, and AM Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

2 Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2008.

3 Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. A foundation for flow-based program matching: Using temporal logic and model checking. In *POPL*, pages 114–126. ACM, 2009.

4 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

5 Shigeru Chiba. Javassist-a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 174. ACM, 1998.

6 James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

7 Barthelemy Dagenais and Martin P Robillard. SemDiff: Analysis and recommendation support for API evolution. In *Proceedings of the 31st International Conference on Software Engineering*, pages 599–602. IEEE Computer Society, 2009.

8 Johannes Henkel and Amer Diwan. Catchup! capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering*, pages 274–283. IEEE, 2005.

9 Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

10 Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009.

11 Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler-a framework for load-time transformation of Java class files. In *SCAM*, pages 100–110. IEEE, 2001.

12 David Lacey and Oege de Moor. Imperative program transformation by rewriting. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 52–68, 2001.

13 Maxime Lamothe and Weiyi Shang. Exploring the use of automated API migrating techniques in practice: An experience report on Android. In *15th International Conference on Mining Software Repositories, 2018*, 2018.

14 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX Annual Technical Conference*, pages 601–614, 2018.

15 Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. Swin: Towards type-safe Java program adaptation between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 91–102. ACM, 2015.

16 Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264. ACM, 2018.

17 Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

18 Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 353–363. IEEE Press, 2012.

19 Gilles Muller, Yoann Padioleau, Julia L Lawall, and René Rydhof Hansen. Semantic patches considered helpful. *ACM SIGOPS Operating Systems Review*, 40(3):90–92, 2006.

20 Beevi S Nadera, D Chitraprasad, and Vinod SS Chandra. The varying faces of a program transformation systems. *ACM Inroads*, 3(1):49–55, 2012.

21 Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. A graph-based approach to API usage adaptation. In *OOPSLA*, pages 302–321. ACM, 2010.

22 Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *2010 ACM/IEEE 32nd International Conference on Software Engineering,*, volume 1, pages 205–214. IEEE, 2010.

23 Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

24 Yoann Padioleau, Julia L Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. *Electronic Notes in Theoretical Computer Science*, 166:47–62, 2007.

25 Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

26 Luis R Rodriguez and Julia Lawall. Increasing automation in the backporting of Linux drivers using Coccinelle. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*, pages 132–143. IEEE, 2015.

27 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*, page 13, 1999.

28 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, pages 357–361. Springer, 2001.

29 Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. Transforming programs between APIs with many-to-many mappings. In *30th European Conference on Object-Oriented Programming*, 2016.

30 Shaowei Wang, David Lo, and Xingxiao Jiang. Understanding widespread changes: A taxonomic study. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 5–14. IEEE, 2013.

31 Louis Wasserman. Scalable, example-based refactorings with Refaster. In *Proceedings of the 2013 ACM Workshop on Refactoring Tools*, pages 25–28. ACM, 2013.