RefaFlex: Safer Refactorings for Reflective Java Programs

Andreas Thies
Lehrgebiet Programmiersysteme
Fernunversität in Hagen
Hagen, Germany
andreas.thies@fernuni-hagen.de

Eric Bodden
Secure Software Engineering Group
EC SPRIDE, Technische Universität Darmstadt
Darmstadt, Germany
bodden@acm.org

ABSTRACT

If programs access types and members through reflection, refactoring tools cannot guarantee that refactorings on those programs are behavior preserving. Refactoring approaches for highly reflective languages like Smalltalk therefore check behavior preservation using regression testing.

In this paper we propose RefaFlex, a novel and more defensive approach towards the refactoring of reflective (Java) programs. RefaFlex uses a dynamic program analysis to log reflective calls during test runs and then uses this information to proactively prevent the programmer from executing refactorings that could otherwise alter the program's behavior. This makes re-running test cases obsolete: when a refactoring is permitted, tests passing originally are guaranteed to pass for the refactored program as well. In some cases, we further re-write reflective calls, permitting refactorings that would otherwise have to be rejected.

We have implemented RefaFlex as an open source Eclipse plugin and offer extensions for six Eclipse refactoring tools addressing naming, typing, and accessibility issues. Our evaluation with 21,524 refactoring runs on three open source programs shows that our tool successfully prevents 1,358 non-behaviour-preserving refactorings which the plain Eclipse refactorings would have incorrectly permitted.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Languages, Reliability

Keywords

Refactoring, reflection, testing

ISSTA '12 Minneapolis, Minnesota USA

1. INTRODUCTION

A refactoring is a behavior-preserving program transformation, typically aimed at improving the program's design [19, 26]. In this paper, we call a transformation a *non-refactoring* if it is intended to preserve the program's behavior but actually does not. Deciding manually whether or not a transformation is a refactoring is hard. Programmers need to take into account many semantic constraints to assure that the transformation, when being applied to a particular program, is indeed behavior preserving [20, 31, 32, 36, 37]. Researchers have therefore developed a range of tools that statically analyze a program's code to proactively prevent the programmer from applying non-refactorings.

Static code analysis must be conservative, however, if the program accesses program elements through reflection. In this paper, we consider refactoring reflective programs in statically typed languages such as Java. Reflection is abundant in Java programs. In previous work [13, 14], we investigated 22 diverse Java programs (drawn from the DaCapo benchmark suite [12] and other sources), all of which turned out to use reflection, ranging from as few as 12 reflective call sites (avrora) to as many as 2809 (tradebeans). In this work we show that reflective calls are a threat to the validity of refactorings. Existing Java refactoring tools ignore reflection entirely, and thus falsely flag non-refactorings as valid refactorings in many cases (1,358 cases out of 21,524 in our benchmark set).

Our approach draws inspiration from the refactoring support for the highly reflective programming language Smalltalk [18]. In this setting, because Smalltalk programs typically use reflection extensively [15], tools like the Refactoring Browser use regression testing to assure that refactorings indeed do not alter the program's behavior [28]. When the refactoring browser discovers, at runtime, a reflective call to a type or member that was previously moved or renamed by a refactoring then this call is rerouted to the new target.

For the setting of Java we propose RefaFlex, a novel approach tuned towards safer refactorings of reflective Java programs. RefaFlex first instruments the program's code base such that the program logs calls to the Java reflection interface during test runs. RefaFlex uses the logged data to generate constraints that express the conditions the refactoring has to obey to preserve the program's behavior. RefaFlex then passes the generated constraints to a satisfiability solver. If the constraint system can be solved, the code transformation is permitted. In this case, re-running test cases becomes obsolete; all test cases passing before transformation are guaranteed to still pass after the transfor-

mation was successfully applied. In case of an unsolved constraint system, there are unfulfilled conditions and the refactoring must not be performed. In some cases, RefaFlex even goes one step further: it uses the constraint system to enhance the refactoring on the fly, re-writing arguments to reflective calls such that the refactoring becomes valid, i.e., such that the enhanced refactoring will preserve the program's behavior although the original refactoring would not have. Interestingly, though, we found and describe cases where behavior preservation may actually be unintended. In those cases, RefaFlex warns the programmer but allows the behavior changes, as they probably reflect the programmer's intention when invoking the refactoring.

Our implementation of RefaFlex seamlessly integrates with the Eclipse IDE [4]. RefaFlex is implemented as a pair of Eclipse plugins that extend six refactoring tools for Eclipse to correctly handle class and member accesses through Java's reflection API. The tools address naming as well as typing and accessibility issues in Java. Our implementation builds on the RefaCola [35], a domain-specific language that allows researchers to formally define constraint generation rules. From those rules, a compiler automatically generates a ready-to-use constraint generator and solver to be used in refactoring tools such as ours.

Current approaches for refactoring tool implementations can guarantee that refactorings will not break existing test cases for programs that do not use reflection. With RefaFlex, programmers obtain the same guarantee also for reflective Java programs. This is because a refactoring, if behavior modification can be ruled out, is limited to changing a program's structure, and reflection is the only way in which the program's behavior can depend on this structure.

To validate our approach, we applied RefaFlex to three publicly available open-source programs known to make use of Java's reflection API. By comparing the results of our reflection-aware precondition checks with the results of the original Eclipse refactoring tools we found that, in our benchmark set, our extensions successfully prevent 1,358 transformations (out of 21,524 in total) that otherwise would have changed program behavior due to reflection.

To summarize, this paper presents the following original contributions:

- a translation of uses of Java's reflection API to equivalent RefaCola constraint generation rules,
- a mechanism preventing non-refactorings caused by reflective accesses to classes and members,
- a constraint-based resolution mechanism to make refactorings valid that would otherwise have to be rejected, and
- a full open-source implementation, along with an evaluation that shows that our approach successfully works on real world programs.

Our implementation, all our RefaCola rules, along with all raw experimental data, and explanations of interesting corner cases are available online at: `http://www.feu.de/ps/prjs/rf/`.

2. EXAMPLE

We next give a general overview of Java's reflection facilities. Then we explain in more detail the problems that may arise when refactoring programs that use Java reflection.

```
1 public class C {
2   public String i = ...  }
3
4 public class Reflection {
5   public void m() throws Exception {
6     Class c = Class.forName("C");
7     Field f = c.getField("i");
8   }
9 }
```

Listing 1: Example program

2.1 Java Reflection

Java allows programmers to query the structure of a running program through a reflection API [24]. The entry points to the reflection API are objects of the class `java.lang.Class`. The programmer can retrieve such a class object, e.g., an object `fooClass` representing a class `Foo`, either through a static method call `Class.forName("Foo")` or through the class constant `Foo.class`.

The programmer can then query `Foo`'s members by using various methods of the class object. The API distinguishes between members declared in the class itself and members that the class inherits from its super classes or interfaces. For instance, calling `fooClass.getMethods()` returns objects representing all public methods declared in `Foo` itself or inherited from one of its super types. A call to `fooClass.getDeclaredMethods()`, on the other hand, only returns those methods that `Foo` itself declares. However, in contrast to `getMethods()`, `getDeclaredMethods()` also returns methods that are private, protected and package-visible. Once a member, let's say a method object `barMethod` representing a method `bar()`, has been retrieved that way, the programmer can interact with that member. For instance, the programmer may call `barMethod.invoke(o)` to invoke the method `o.bar()` on a receiver object `o`. For methods with parameters, the `invoke` method allows a variable number of additional arguments that again may be obtained through reflection. In case `bar()` is private, a call to `barMethod.invoke(o)` will typically fail with an `IllegalAccessException`. Depending on the security settings of the running virtual machine, a programmer can obtain invocation rights to private methods by calling `barMethod.setAccessible(true)`. Besides method accesses, the reflection API also supports constructor and field accesses in analogous ways. For the latter, the following section will give an example.

2.2 Refactoring reflective Java programs

As we explained above, programmers can use reflection to query their program's structure. A refactoring can change this structure, hence invalidating those queries if no special care is taken. For instance, consider the example program in Listing 1. In this example, the class `Reflection` accesses the field `C.i` through reflection.

One of the most common invariants for behavior preserving program transformations is to guarantee that name bindings within the program do not change when the program is executed [32] (up to some well-controlled changes that actually make up the intended transformation). Let us assume that the programmer applies a refactoring renaming `C.i` to `C.j`. In the program from Listing 1, if `C.i` were accessed through a direct field reference in the source code, the refac-

```
1  public class Super {
2    public int j = ...  }
3
4  public class C extends Super {
5    public int i = ...  }
6
7  public class Reflection {
8    public void m() throws Exception {
9      Class c = Class.forName("C");
10     Field j = c.getField("j");
11     C o = (C) c.newInstance();
12     j.set(o, 42);
13   }
14 }
```

Listing 2: Example program with class hierarchy

toring would have one of two options to make sure that name bindings do not break: (1) reject the refactoring, leaving the source code untouched, or, more likely to be chosen, (2) rename any references to `C.i` such that they refer to `C.j` after the refactoring has completed.

When programs access classes or members through reflection, such as in Listing 1, things are not quite as simple. By just considering the source code of the given program it is not straightforward to determine that it accesses `C.i`, since those accesses happen through a combination of multiple method calls (lines 6–7). In the example, the situation is not hopeless, though, because the strings used to compute the field reference `C.i` are present in the program's code. In such situations, static string analyses can be effective in determining the target of a reflective method call [17, 23]. In many cases, however, the parameters used in calls to the reflection API are not string constants but rather are loaded from configuration files unknown to the refactoring tool [14, 29]. Without additional domain specific knowledge, this makes it virtually impossible for a refactoring tool to update those values automatically.

The current state of the art in refactoring Java programs is that reflective calls are ignored. For the example in Listing 1 this means that the rename-field refactoring would complete without warning, and simply ignore the reflective field access. When the resulting program then executes, this will cause a `NoSuchFieldException` to be thrown. Thus, such a transformation, although allowed by current tools, is clearly not behavior preserving.

2.2.1 *Altered bindings through inheritance*

In the above case, the transformed program at least fails with an exception, making the programmer aware of the broken name binding at runtime. It is easy to construct examples, though, where the transformed program executes without exception but nevertheless not as originally intended.

Consider the program in Listing 2. Here the program accessed the field `Super.j` through a reflective access on the type `C`. This look-up succeeds because the method `getField(..)`, just as `getMethod(..)`, searches the receiver's complete type hierarchy until a field is found that matches the arguments passed to the `getField(..)` call.

Next, consider the rename-field refactoring as before, renaming `C.i` to `C.j`. In the hierarchy resulting from applying this refactoring, the same field look-up of `C.j` will now bind to the renamed field `C.j`, not `Super.j`. Such an unintentionally modified binding may change the program's behavior and, because it goes without any runtime exception, may well lead to subtle bugs.

2.2.2 *Visibility changes*

Not only can altered name bindings break reflection code, the same is true for refactorings that modify the visibility of a class or member. The reflective field assignment in line 12 of Listing 2 succeeds (in the original, un-refactored program) only because the field `Super.j` and its declaring type are public. Next, consider a refactoring that removes `Super`'s `public` modifier, e.g., in the context of an Extract Interface refactoring. In the resulting program, the `set`-call at line 12 would fail with an `IllegalAcccessException`. In this case, the problem would be minor, though. The programmer could alter the program, adding a call `j.setAccessible(true)` just before line 12. This will cause the reflection API to disregard access restrictions on the field, allowing the field assignment to succeed. (Whether or not `setAccessible` may be called depends on the security settings of the running virtual machine. Such calls are allowed by default.)

Calls to `setAccessible` do not, however, alter the semantics of method calls such as `c.getField()`. The `getField` method will return only such fields that are declared as `public`. While a call `setAccessible(true)` makes a field accessible, it does *not* change its modifier to `public`, and hence the call to `c.getField("j")` in line 10 will fail with a `NoSuchFieldException` if a refactoring would remove the `public` modifier of `j`. In this case, the programmer could rectify the problem by instead using `Super.class.getDeclaredField("j")`. Opposed to `getField`, the method `getDeclaredField` also returns such fields not declared as `public`. However, unlike `getField`, it does not search the type hierarchy.

2.3 Discussion

As the reader may have noticed, the possible effects of refactorings on reflective Java programs are anything but trivial to foresee. Many code transformations may need to be disallowed because reflective calls would cause these transformations to be not behavior preserving. In some cases, however, one can design additional code transformations on the reflective code itself that, when combined with the refactoring transformation, will cause the combined transformation to preserve the program's behavior after all.

In the remainder of this paper we propose a solution that (1) records information about how a program uses reflection at runtime, (2) uses refactoring constraints based on this information to prevent semantics-changing code transformations, and in some cases (3) allows the rewriting of code that uses the reflection API such that the rewritten code will exhibit the same behavior as the original program for the given test cases.

3. GENERATING REFLECTION USAGE CONSTRAINTS

Traditional refactoring specifications usually consist of two parts, a set of preconditions and a specification of the mechanics, i.e., an algorithmic part describing the necessary code transformations [19]. The preconditions are checked before the refactoring, such that the mechanics may assume the preconditions to guarantee valid output.

Constraint-based refactoring [37] is a way to implement refactoring tools that softens the strict distinction of pre-

condition checking and mechanics by combining both in a single constraint-solving pass in which the preconditions are formulated as constraints connected with each other through shared variables. The variables themselves represent the changes the refactoring plans to perform. The variables' domains match the allowed changes the refactoring tool might perform. Once the constraints are generated, a constraint solver attempts to fulfill the constraints by assigning proper values to the variables.

To give an example: From a constraint-based view, renaming a method limits the domain of possible names for this method to what the user specified as the method's new name. The fact that the new name must be propagated to all the method's references is expressed in terms of constraints, forcing the references to be named as the method. Hence, the constraint solver will try to rename the references as well. Doing this, further constraints may apply, e.g., that a reference must not be renamed if it will then bind to another (overloading) method. Depending on the existence of such overloading methods, the constraints are either solvable or not. If a constraint system turns out to be unsolvable, the corresponding refactoring must not be performed. For a solvable constraint system, each solution directly holds instructions for the necessary code transformations. Likewise, for constraint systems with multiple solutions, there are multiple code transformations that solve the corresponding refactoring problem. Usually, in this case a user would prefer the refactoring resulting in the smallest number of changes in total. (Finding an optimal refactoring among a set of possible solutions is an interesting research problem but out of the scope of this paper.)

We base our notation and implementation of RefaFlex on this constrained-based approach for reasons of flexibility. Reflection may affect almost all kinds of Java refactorings. A constraint-based approach promises to be very generic because its formulation is independent of a specific refactoring problem. A concrete refactoring is encoded as a set of constraint variables and our approach works as long as the set of offered variables is expressive enough to encode the refactoring's parameters.

3.1 Constraint Variables

The usefulness of a constraint-based approach directly relates to what kinds of constraint variables are considered. We incorporate variables for all program properties typically modified by standard refactorings, such as identifiers, accessibility modifiers, types, and locations. Figure 1 lists all variables we consider, including their domains.

Given the different variables from Figure 1, it becomes clear that even very small programs may be equipped with a manifold set of constraint variables. Take for example Listing 1, which induces more than a dozen of variables. Both type declarations, that of class `C` and class `Reflection` have associated variables *identifier*(`C`) and *identifier*(`Reflection`). A Rename type refactoring can then be formulated by changing the value of one of these variables. Both types further have associated variables *package*(`C`) and *package*(`Reflection`), representing their current enclosing package (the default package in the given example) as well as variables *accessibility*(`C`) and *accessibility*(`Reflection`) representing their accessibility (which is currently `public`). Changing one of their values will represent a Move type refactoring or a change of accessibility, respectively.

Same as the type declarations, also the field and method declarations of Listing 1 are equipped with variables representing their names, parameter types, and host types to represent various other refactorings such as Pull up field or Change method signature.

For calls $r \in R$ to the reflection API, Figure 1 defines four additional constraint variables. For line 12 of Listing 2, for example, *package*(`j.set(o,42)`) and *type*(`j.set(o,42)`) specify the package and type in which the reflective call resides. This permits not only refactorings modifying reflectively-referenced declarations but also refactorings that move the reflective code itself. The variable *isAccessible*(`j.set(o,42)`) denotes the state of the receiver `j`'s `isAccessible` flag at this call site. At line 12 our dynamic analysis infers that this value is `false` (by evaluating `j.isAccessible()`).

Finally, for the call at line 10 of Listing 2, the variable *stringArg*(`c.getField("j")`) refers to the string argument passed to the reflective method. Our dynamic analysis infers the variable's current value `"j"`, no matter whether it is evident from the source code or computed by an arbitrary expression. In case a call to the reflection API is invoked more than once during program execution (with potential differing arguments) RefaFlex generates one variable for each reflective invocation.

3.2 Constraint rules

While constraint variables express the potential degrees of freedom for a certain refactoring, the constraints are their counterpart guarding over all necessary restrictions a refactoring has to obey. Constraints are generated from patterns, the so called *constraint generation rules*.

Our constraint rules for correct handling of reflection are inferred from the Java API specification [1]. Due to space restrictions, this paper can only outline the set of constraint rules required to refactor reflective code. For the interested reader, we have made a full specification of all reflection rules publicly available at our project website.

Our constraint rules use the following syntax:

$$\frac{\textit{program query}}{\textit{constraint}}$$

where *program query* represents a Datalog-like [16] program query and the *constraint* shows a pattern according to which constraints are generated for each variable assignment matched by the query. Queries are evaluated based on both dynamic log files and the program's static structure.

Naming.

When renaming declarations, one must consider calls to the reflection API expecting or returning the declaration's name. As outlined above, renaming the field `C.i` in Listing 1 without considering the reflective access in line 7 results in a `NoSuchFieldException`. The following constraint generation rule avoids this problem:

$$\frac{\textit{holdsName}(r, d)}{\textit{stringArg}(r) = \textit{identifier}(d)}$$

where $\textit{holdsName}(r, d)$ indicates that the reflective call r receives or returns the name of declaration d as a string. For the reflective access in line 7 (Lst. 1) this rule produces:

stringArg(`c.getField("i")`) = *identifier*(`C.i`)

assuring that the passed string will match the declaration's name past the refactoring. Note, that a change of the right-hand side of the constraint (a rename of `C.i`) not necessarily

Variables:

$identifier(d) \in String$	the identifier of d
$accessibility(a) \in Acc$	the access modifier of a
$package(t) \in P$	the declaring package of t
$type(m) \in T$	the declaring type of m
$parameters(m) \in T^{n_m}$	m's n_m formal parameter types
$stringArg(r) \in String$	the string passed or received by r
$isAccessible(r) \in boolean$	r's `isAccessible()` value
$package(r) \in P$	the package in which r resides
$type(r) \in T$	the type in which r resides

with $d \in D$, $t \in T$, $m \in M$, $a \in A$ and $r \in R$

Domains:

P	the set of declared packages
T	the set of declared types
M	the set of decl. methods, fields and constructors
D	$= P \cup T \cup M$
A	$= T \cup M$
Acc	{private, package, protected, public}
R	the set of calls to the reflection API

Figure 1: Constraint variables and their domains

leads to an unsolvable constraint system. As we will see in Section 4, we can also offer program transformations rewriting reflective statements. This allows the constraint solver to also solve constraints of this kind.

Scoping.

When moving declarations during a refactoring, one must take care not to move a referenced declaration outside the scope through which it is referenced: the invocation of `Class.getField` in line 10 of Listing 2 will only return the field `j` as long as it is declared in `C` or one of its super types. The following rule assures that `j` remains in scope:

$$\frac{Class\#get^*(r, m)}{receiver(r) \leq_T type(m)}$$

Here, the query $Class\#get^*(r, m)$ expresses that r is of type `Class.getField`, `getMethod` or `getConstructor`, and r accesses the member or constructor m. We obtain results for this query by evaluating it against the collected runtime information. The function $receiver(r)$ evaluates to the class on which the method was invoked and $\leq_T$ refers to the subtype relationship.

Accessibility.

Many methods within the Java reflection API take into account accessibility restrictions. Reflection API calls such as `Class.getField` only return members declared **`public`**. The according rule can be formulated easily as follows.

$$\frac{Class\#get^*(r, m)}{accessibility(m) = \texttt{public}}$$

This rule demands that elements previously accessed through `Class.getField`, `getMethod` or `getConstructor` (as evident by the collected runtime information) remain publicly accessible. A more sophisticated rule is required for methods such as `Field.get/set` and `Method.invoke`, the reflective counterpart for field accesses and method invocations. These accesses require either the `setAccessible`-flag of the referenced declaration to be set to **`true`** or require both the accessed declaration and it's enclosing class to be accessible. We can express this as:

$$\frac{accesses(r, m)}{\begin{array}{c}(accessibility(m) \geq_A minAcc(type(r), type(m)) \\ \wedge\ accessibility(type(m)) \geq_A minAcc(type(r), type(m))) \\ \vee\ isAccessible(r) = \texttt{true}\end{array}}$$

where we define a helper function $minAcc(t_1, t_2)$ as

$$minAcc(t_1, t_2) = \begin{cases} \texttt{private} & \text{if } t_1 = t_2 \\ \texttt{package} & \text{else, if } package(t_1) = package(t_2) \\ \texttt{public} & \text{else} \end{cases}$$

to calculate whether a declaration in t_2 is reflectively accessible from t_1. Note how this function differs from its equivalent for regular accesses [33]:

$$\alpha(t_1, t_2) = \begin{cases} \texttt{private} & \text{if } topleveltype(t_1) = topleveltype(t_2) \\ \texttt{package} & \text{else if } package(t_1) = package(t_2) \\ \texttt{protected} & \text{else if } \exists t \in T : type(t_1) \leq_N t <_T type(t_2) \\ \texttt{public} & \text{otherwise} \end{cases}$$

While – for regular accesses – the accessibility of private members depends on their outermost (top-level) type, the reflection API only considers the innermost enclosing type. Also, Java reflection does not at all consider **`protected`** accessibility or nested types (expressed by $\leq_N$).

Hiding.

Things become even more interesting when hiding comes into play. In Figure 2, we showed an example in which renaming a declaration would change a reflective binding due to hiding. The following rule forbids such changes:

$$\frac{Class\#get^*(r, m), member(m')}{\begin{array}{c}identifier(m) \neq identifier(m')\ \vee\ type(m) <_T type(m') \\ \vee\ type(m') <_T receiver(r)\ \vee\ accessibility(m') <_A \texttt{public} \\ \vee\ parameters(m) \neq parameters(m')\end{array}}$$

When accessing a member m, there must not be any other competing member m' in the type hierarchy of m with matching name, matching parameter types, and sufficient accessibility to which the access r could accidentally bind past a refactoring.

Note how the declarative formulation of this constraint rule not only allows to prevent accidental hiding but also gives a constraint solver the opportunity to solve the constraint by changing other variable's values. For the example from Figure 2, a rename of `C.i` to `j` remains feasible by lowering the accessibility of `i`. Likewise, when a member is moved to a class where it henceforth will hide another declaration, the constraints still allow renaming the moved declaration. Upcoming ripple effects due to this renaming are handled by other constraints as well. Wherever a reference points to the renamed declaration, the name in the reference will be updated thanks to the above naming rule.

Once the constraints for a specific refactoring problem have been generated, RefaFlex passes them to a satisfiability solver. The solver's outcome is either the information that the refactoring cannot be performed due to unsatisfi-

able preconditions/constraints, or it is a set of changed variables, each representing a single change in the program. In many cases, a constraint system might have more than one solution, which means that there are multiple code transformations meeting all conditions. Often, this might be due to a spare `setAccessible(true)` invocation (see below) where a declaration is already accessible. We currently just use the first solution found by the constraint solver, which is typically the solution with the fewest changed variables.

4. REFACTORING CALLS TO THE REFLECTION API ITSELF

As all variables belong to Java declarations, RefaFlex can transform the code in a straightforward fashion. For example, if a variable *identifier*(d) representing a declaration's name changes its value, RefaFlex will rename d to the value of *identifier*(d). Similarly, a change of *accessibility*(d) means that the accessibility of d must be set to the new value of *accessibility*(d) by replacement, insertion or deletion of access modifiers.

It is less obvious, however, how to perform necessary changes to reflective API calls. Figure 1 defines four different kinds of variables for a reflective invocation r. *type*(r) and *package*(r) specify the location of r in terms of its enclosing type and package. Both change their value if and only if an enclosing declaration is moved, such that these changes are performed implicitly along with the movement of the parent container. Changes to *isAccessible*(r), however, referring to the `setAccessible` flag for a reflective invocation, and *stringArg*(r), expressing a declaration's name used inside a reflective call, induce explicit code changes.

Given an invocation of `Field#get(Object)`

```
... <expression>.get(<parameter>);
```

for which the constraint solver requires the `setAccessible` flag set to `true` (e.g., because the field referenced by `<expression>` gets inaccessible due to a "move" refactoring) the `setAccessible` flag can be set by modifying the code according to the following pattern:

```
Field <freshName> = <expression>;
<freshName>.setAccessible(true);
... <freshName>.get(<parameter>);
```

Here, `<freshName>` represents a fresh local variable introduced to avoid side effects caused by multiple invocations of `<expression>`. In rare cases, additional effort has to be taken. For instance, we have to generate a helper method when the reflective invocation resides inside a super-constructor invocation, as this invocation must remain the first statement inside the constructor's body.

If the constraint solver assigns a new value to a variable *identifier*(r), i.e., a name change in a reflective invocation r (because the referenced declaration was renamed), we distinguish two cases, depending on whether the changed name flowed into the reflection API call as a parameter or was returned from a reflection API call. In the case of parameters, to guarantee that an invocation of such a method still returns the same declaration after renaming, the passed string must be replaced. In general, however, that string might be computed by an arbitrary expression, e.g., loading the string from a file, which precludes us from replacing the string's value in the expression itself. Nevertheless, we can perform changes to the code to make it pass the given test cases:

```
String <freshName> = <expression>;
// TODO: consider renaming of <old name>
//       to <new name> in <expression>
if(<freshName>.equals("<old name>")
   <freshName> = "<new name>";
... Class.forName(<freshName>)
```

That way, the refactoring fixes the problem for the tested inputs. Through the included `TODO` comment, we remind the programmer to propagate the renaming inside the `<expression>` manually.

Intentionally granted modifications of behavior.

For reflective invocations *returning* a declaration's name, we decided not to provide automatic code transformations. Even though the pattern above could have been adopted easily, we noticed that API calls such as `Method#toGenericString()` or `Field#getName()` are commonly used for debugging, display and logging purposes. In this case, it makes little sense to still refer to an old name past a refactoring. By ignoring such calls to the reflection API, the resulting transformation will not be strictly behavior preserving in the sense that it will display or log the changed names after the transformation. However, it *will* be behavior preserving in that the resulting program will remain to correctly display or log the (changed) name of the same program element. Hence, in this case RefaFlex just provides appropriate warnings.

5. IMPLEMENTATION

Section 3 and 4 described in abstract terms how to determine and fulfill reflection constraints, i.e., the constraints that a refactoring must obey to ensure a behavior-preserving transformation in the presence of reflective calls. In this section, we report on our concrete implementation, an extension to six refactoring tools for Java. Our prototype RefaFlex is integrated into the Eclipse IDE, as a pair of Eclipse plugins, one RefaCola plugin taking care of the refactoring support and one TamiFlex plugin for gathering information about calls to Java's reflection API. In combination, both plugins seamlessly integrate the RefaFlex methodology into the Eclipse IDE: When programmers test-run their programs within the IDE, RefaFlex gathers runtime information about reflective calls. When programmers then invoke a refactoring, RefaFlex causes the reflection rules from Sections 3 and 4 to be instantiated, with queries being evaluated based on the gathered information. In result, RefaFlex guarantees that the same test runs executed initially will succeed also past refactoring.

In previous work, the second author developed the TamiFlex tool chain [13, 14] to improve the static analysis of Java programs that use reflection. TamiFlex consists of three components, but for this work we only use one of them: the Play-out Agent, a Java agent that instruments Java's reflection API at load time, such that any calls to this API will be logged into a file on disk, in our case into the Eclipse workspace. The instrumentation occurs within the Java runtime library itself, which makes the instrumentation both lightweight and generic: TamiFlex logs all calls to the reflection API, including such that are themselves issued through reflection. The Eclipse plugin for TamiFlex allows users to run their application in a special run mode (similar to RUN/DEBUG) that runs the application or JUnit test cases with the Play-out Agent attached. By default, TamiFlex writes log files in an accumulative manner: when the user

triggers multiple different test runs then the log is expanded with the data of each run. In previous work we could show that this process will usually result in stable log files if the test cases cover all of the program's features. In particular, fine-grained path coverage is typically not required. At any time, the plugin exposes a list of log files gathered for the benefit of other plugins, such as the RefaCola plugin in our case.

For the implementation of the reflection constraints we made use of the RefaCola [35], a domain specific language for the formal specification of constraint rules. The RefaCola comes with a compiler that automatically translates refactoring specifications into a ready-to-use constraint generator and solver, which can easily be used within refactoring-tool implementations for various IDEs such as Eclipse. Besides that, RefaCola offers solutions for most of the technical problems of constraint-based refactoring, e.g., by providing techniques to only generate those constraints actually necessary for the intended refactoring [35].

We implemented our refactoring engine as an extension to traditional refactoring tools, which use a static analysis for their precondition checks but are blind for reflective behavior. Clients are allowed to register their desired changes to this extension using pairs of variables (as defined in Figure 1) and their desired values. The diversity of the offered variables currently allows us to express most standard refactorings that rename, move and hide declarations and that change declared types. Based on this, we extended six refactoring tools of the Java Development Tools (JDT) [5], namely Rename Field, Rename Method, Rename Type, Rename Package, Move Type, and Change Method Signature.

This subset of six out of approximately 30 refactoring tools in Eclipse is not chosen at random, but for technical reasons. Eclipse currently does not offer extension points for other than these six tools,[1] making it impossible to participate in the refactoring process through the official API. If such extension points were present, our approach could be applied to a much broader range of refactorings moving or introducing program elements such as Pull up field or Extract method, since RefaFlex's constraint-based formulation is independent of a specific refactoring problem.

Limitations

Our approach and implementation have some limitations which we next discuss to allow for an unbiased presentation, and to allow other researchers and practitioners to build on top of our approach and implementation. We believe that those limitations are not nearly severe enough to prevent RefaFlex from being useful in practice.

Even though we offer program transformations to fix accessibility and naming issues, there are cases in which the constraint system will remain unsolved. This can happen when a refactoring were to pull a declaration outside the scope from which it is referenced. Also, RefaFlex cannot offer appropriate program transformations if a reflective access comes from a non-editable library. JUnit, for example, collects test methods through reflection. Moving such a method will cause RefaFlex to show an error message pointing the user to the reflective access. The library call within JUnit cannot be rewritten.

[1]See Eclipse bugs 86438 and 89422

Our approach shares an inherent limitation with other refactoring tools based on dynamic analysis, like the Smalltalk Refactoring Browser [28]: the approach can guarantee refactoring correctness only with respect to the reflective method calls that were dynamically recorded. While a true limitation, we do not believe that it prevents the approach from being useful in practice. Many Java developers not only develop but also test-run their applications in integrated development environments (IDEs) like Eclipse, and our solution integrates with Eclipse, which allows those developers to collect log files without any additional effort. Furthermore, it is hard to conceive *any* general solution to the problem of reflection. Reflective method calls can draw their parameter values from any kind of program inputs. While current IDEs do offer support for recognizing (and even refactoring) a restricted set of identifiers for a restricted set of types of configuration files, this approach also can never be a general solution, as file types may generally vary greatly. Our approach is a pragmatic, partial, best-effort solution that means to complement existing tool support.

RefaFlex can replace existing refactoring support for configuration files to some extent but not entirely. When a configuration file defines a string value which then ends up being used in a reflective method call, RefaFlex will pick up that string value during runtime, and hence will successfully prevent the call's target from being renamed. On the other hand, RefaFlex has no way to trace the string back to its origin, and therefore cannot offer the programmer a refactoring that would rename the string at its point of definition. As we showed in Section 4, RefaFlex will provide a hint to the user in such situations, in the form of a `TODO` comment. RefaFlex truly excels when being combined with other existing refactoring tools. Since RefaFlex integrates with all the refactoring tools implemented in Eclipse, programmers can easily make use of those synergies.

A further limitation of RefaFlex is that, as a generic approach, it is not aware of any program or framework specific restrictions. In the case of JUnit, for example, it will be unproblematic to rename a method from `testFoo` to `testBar`, as JUnit treats all methods with the prefix `test` equally. RefaFlex, however, does not have this information, and hence would prevent such a rename refactoring. This is safe but not as flexible as one may wish.

One minor limitation of RefaFlex is that it only treats situations in which a reflective method call succeeds before refactoring, and would break after. The inverse is, at least in theory, also imaginable. A program may contain an unsuccessful reflective method call. RefaFlex currently does not monitor such calls. A refactoring could, however, just by chance, happen to rename a type or member in exactly such a way that the reflective call successfully binds to this type or member after the renaming has been applied. RefaFlex could be extended to treat such cases as well, by monitoring and treating successful and unsuccessful calls separately, however we believe the general problem to be rather theoretic and hence did not implement this solution.

A final minor limitation is that those code transformations of ours that introduce `setAccessible(true)` will only work for programs that either have no security manager installed, or in which the security manager allows accessibility rights to be granted. This could be rectified by having RefaFlex probe the security manager when log data is collected at runtime.

6. EVALUATION

Our evaluation addresses the following research questions:

RQ1 (Necessity): What fraction of refactorings would alter program behavior due to reflection if reflection were ignored in the refactoring process?

RQ2 (Flexibility): What additional flexibility does one obtain by allowing rewrites of reflective call sites?

RQ3 (Correctness): Does RefaFlex manage to treat all the above cases correctly?

RQ4 (Efficiency): What additional runtime cost does the approach impose on the development cycle?

The following subsections address those questions separately. Section 6.5 discusses threats to the validity of our experiments.

6.1 RQ1: Necessity

To indicate whether our approach gives solution to a relevant problem, we conducted a series of experiments in which we applied a total of 21,524 refactorings systematically to three benchmark projects known to use reflection within a total of more than 1,200 classes: Play, Joda and JCC. All projects are publicly available [2, 7, 8].

We chose those subject programs because their source code is easily accessible and because all projects come with a decent set of test cases. As explained earlier, test cases are necessary for our approach. For each project, we first used the RefaFlex Eclipse plugins to execute the respective test case, yielding a log file filled with information on all reflective accesses occurring on those test runs. The projects and test cases are available on our website, along with our implementation (in source) and detailed documentation on how to produce our results.

Next, we attempted to apply the six refactorings Rename Field, Rename Method, Rename Type, Rename Package, Move Type and Change Method Signature systematically to all program locations at which they could potentially be applied. As input parameters for the Rename refactorings we have chosen fresh names currently unused in the given project, for Move Type we used arbitrary (but already existing) packages as destinations. Change Method Signature was instructed to declare a method as `private` if it was at least package accessible and to make it `public` if it were declared `private`. Our implementation can handle other changes to method signatures as well. Including those additional degrees of freedom in our evaluation would have allowed too many possible refactorings, though, to be able to finish the evaluation in a reasonable time frame. Moreover, our handling of type correctness and name binding is already validated through the other refactorings.

We repeated each refactoring attempt three times, using different configurations. Table 1 summarizes the results of our evaluation. The first column states the name of the benchmark project used. Then, for each refactoring there follow four columns that summarize our empirical results for applying this refactoring to the respective project.

First, we give the total number of refactoring attempts conducted in each of the three configurations, "total". Since we applied the refactorings at every place they could be applied, the "total" numbers for the various Rename refactorings and Change Signature also express how many declarations of these kinds reside in the benchmark programs. The "total" numbers for Rename Type differ from those for Move Type because Eclipse does support the renaming but not the moving of inner nested types.

As stated above, we attempted each refactoring in three configurations of the refactoring tools. In this section we focus on the first two configurations. First we applied the JDT refactorings as they are. The column labeled E shows how many refactorings Eclipse rejected execution due to preconditions that were unfulfilled even without taking reflection into account. In a second run, we activated RefaFlex with the additional RefaFlex precondition checks to handle reflection but did not allow it to make changes to reflective invocations. In column R, we show the number of refactorings for which the preconditions failed with RefaFlex's constraint checker enabled.

As our results show, for each refactoring there are a number of cases (calculated by subtracting column E from R, 1,358 cases in total), in which Eclipse would have falsely allowed a code transformation to be executed, although it would move or rename a type or member that is actively referenced by a reflective method call. Test-running any of those transformed programs results in an altered program state immediately after the reflective access, a "weak" mutation [25], potentially propagating through the whole program up to the program's output. We observed failing tests in many such cases.

6.2 RQ2: Flexibility

The R+ columns in Table 1 show the results of applying the refactoring with our third configuration, in which we unleashed our approach's full power by also enabling rewrites to reflective invocations by allowing the constraint solver to change values of *isAccessible* and *identifier* variables of reflective invocations as described in Section 4.

As our results show, for Rename Field and Rename Method there are a total of 178 cases in which reflective statements were successfully rewritten to make a refactoring behavior preserving, whereas the original refactoring would have been forbidden by RefaFlex, as it would have changed the program's behavior. (Subtract column R+ from R.) In the remaining cases, RefaFlex oftentimes failed to offer additional changes, usually because the reflective invocations originate from within binary libraries. We believe that this reflects what can be expected in practice. Reflective code often comes in terms of libraries, e.g., in the context of test- or dependency injections frameworks. The strength of RefaFlex then results from its error messages, notifying the programmer that, for instance, a testing framework might treat a renamed method different past refactoring. We conclude that there is a significant number of cases in which our reflection rewriting rules are useful.

6.3 RQ3: Correctness

In the previous sections we claimed that RefaFlex would give programmers the guarantee that test cases executed before refactoring will still pass after refactoring, even if reflection is involved. To validate this claim, we first checked that all program variants produced by applying RefaFlex still compile, and for a limited set of variants (about 10% of the

	Rename Field				Rename Method				Rename Type				Rename Pkg				Move Type				Change Signature			
Program	total	E	R	R+	total	E	R	R+	total	E	R	R+	total	E	R	R+	total	E	R	R+	total	E	R	R+
Play [8]	1014	8	15	14	2875	457	485	482	521	0	11	11	43	0	9	9	280	0	11	11	2875	878	907	907
Joda [2]	58	0	0	0	236	28	109	107	67	0	2	2	6	0	1	1	36	0	2	2	236	75	163	163
JCC [7]	878	34	36	34	5661	1358	2193	2023	645	5	5	5	7	0	0	0	425	5	6	6	5661	2525	2776	2776
Σ	1950	42	51	48	8772	1843	2787	2612	1233	5	18	18	56	0	10	10	741	5	19	19	8772	3478	3846	3846

total = no. of refactorings, E = rejected by Eclipse, R = rejected by RefaFlex, R+ = rejected even under possible rewriting

Table 1: Quantitative evaluation for our six refactorings

total 21,524 cases) we also re-ran the test cases that came with the respective project. We calculated that running the test harness for all 21,524 cases would take more than 10 days, which was more time than we had available at our disposal. While running the first few hundred refactorings together with their tests indeed helped us to uncover some incorrect or incomplete refactoring rules, eventually our approach produced no observable abnormalities any longer. We therefore have reason to believe that our current prototype indeed does provide the above guarantee.

6.4 RQ4: Efficiency

An approach like RefaFlex is only useful if it does not unduly disrupt the software development process. As explained before, RefaFlex integrates with Eclipse in a seamless fashion, adding virtually no additional burden to the programmer. In previous work, we found that the runtime overhead during log-file recording is virtually not perceivable [14], except for one pathological program run, tradesoap, which executes 684,486 reflective calls in under one second. But even in this case the overhead is about 160% and thus not necessarily prohibitive. Nevertheless, our Eclipse plugin gives the programmer the explicit choice to execute a run configuration with or without RefaFlex attached.

We tried to measure the execution time of the refactoring applications but due to many influencing factors in the Eclipse IDE the variance was too high to obtain reliable results. Nevertheless, RefaFlex appears to add no noticeable delay to Eclipse's respective built-in refactorings.

To summarize, we have shown that our approach is necessary, offers added flexibility by rewriting reflective call sites, is correct to a reasonable degree, and is efficient enough to allow seamless use in modern IDEs.

6.5 Threats to validity

As any empirical study, the external validity of our experiments is limited by the choice of study subjects. We have chosen programs for which source code and test cases were available, and of which we knew that they used reflection to some degree. Especially given our previous work [14], we believe those programs to be representative of a large class of Java programs. Nevertheless, we cannot claim that RefaFlex would perform equally well when applied to subjects outside our study.

The internal validity is threatened by the fact that we are unable to reliably determine whether or not a refactoring actually is behavior preserving. While we did run test cases, those test cases are certainly incomplete, and do not cover all aspects of behavior preservation. On the other hand, while total correctness is our goal, we only claim that RefaFlex guarantees the passing of test cases that also passed before refactoring, which is indeed the case for all test cases we executed.

7. RELATED WORK

We discuss research on refactoring for reflective programs, research on reflection and static analysis, as well as state-of-the-art refactoring support in current IDEs.

7.1 Refactoring for reflective programs

The only other approach to date that explicitly combines refactoring and reflection is the Refactoring Browser [28] for Smalltalk. As Smalltalk is an inherently reflective programming language, virtually all Smalltalk programs use reflection [18]. The Refactoring Browser thus follows a test-driven approach in which test cases are used to assure the program's correctness after refactoring. In some cases, Refactoring Browser even goes one step further: after renaming a method, for example, if the Refactoring Browser detects a call to the original method during the execution of the (apparently insufficiently) refactored program, the tool rewrites the source code causing that call at runtime [27].

While RefaFlex draws inspiration from Refactoring Browser, it intentionally differs in several respects. First, RefaFlex rewrites source code only at refactoring time, not at runtime. We feel that it is an important invariant to preserve for Java programmers that programs do not have their source code rewritten during their own execution. Second, RefaFlex is based on constraint solving, which provides an automatic and holistic solution to the refactoring problem. If a solution is possible, the constraint solver is guaranteed to find it, and will find this solution automatically. This is different from Refactoring Browser and other tools not based on constraint solving, which must instead hard-code a fixed set of rules directly in their implementation. Last but not least, an important difference between RefaFlex and Refactoring Browser is rooted in the reflection APIs that they deal with. Both APIs differ substantially. The rules derived for RefaFlex could not directly be applied to Smalltalk, and would likely also require at least slight adjustments for other languages similar to Java.

7.2 Static analysis and reflection

To enable static analysis in the presence of reflection, Livshits, Whaley and Lam [23] present a static-analysis approach that infers additional information about reflective call sites directly from program code. The analysis attempts to use information stored in string constants to resolve reflective calls statically. For call sites for which this information is insufficient, their approach allows programmers to provide additional information through manual hints. Christensen et al. present a general-purpose static string analysis [17]. As we observed in previous work [14], many reflective calls

are resolved not using string constants but using information from the environment or configuration files, limiting the utility of such approaches. Our findings are backed up by a study by Sawin and Rountev, who show that in their benchmark set a static string analysis only succeeds in about 28% of all reflective call sites [29]. Nevertheless, such static approaches could be used to generate a subset of RefaFlex's constraints even if no program runs were recorded.

As Sawin and Rountev further show, for any given system one can statically resolve up to 74% of the same reflective call sites by including information about the values of environment variables in that system. As in our case, this information is gathered through a dynamic analysis [29]. Those results reconfirm our design decision to base RefaFlex on dynamically gathered execution information. By recording the parameters to all calls to the reflection API, RefaFlex's log files comprise information about relevant environment variables as well.

Sawin and Rountev further propose a classifications of (generally not conservative but often reasonable) assumptions that a call-graph construction algorithm can make about dynamic features such as reflection [30]. As the authors show, those assumptions can lead to significantly smaller (and hence more precise) call graphs compared to a fully conservative variant.

Hirzel et al. [21,22] present an online version of Andersen's points-to analysis [11] that executes alongside the program, as an extension to the Jikes RVM (formerly Jalapeño) [10], an open-source Java Research Virtual Machine. As an online algorithm, the approach can exploit runtime information; for instance, it can observe reflective calls as they execute.

7.3 Refactoring of non-code artifacts

Modern IDEs have realized the maintenance problems caused by textual references to classes and their members in non-code artifacts such as configuration files. Without further support, programmers could easily break the link between the code and those artifacts, just through simple renaming operations. Therefore many modern IDEs have extended their refactoring support to include non-code artifacts as well. The Plugin-Development Environment [9] in Eclipse, for example, recognizes references to classes that occur in plugin configuration files (`plugin.xml` and `MANIFEST.MF`). When the programmer invokes a rename refactoring on the class, the class is renamed consistently both in the code and in those files. Eclipse furthermore supports a general mechanism that identifies fully qualified class names in any kind of text file. When invoking a refactoring renaming an identifier, such as Rename Class, the user can choose whether to rename fully qualified references in text files as well.

Eclipse [4], as well as the NetBeans IDE [3] have special refactoring support for Spring beans configuration files. The IDEs support renaming such references in those files along with the referenced class/method definitions as part of a refactoring. The Spring refactoring support contributes references from beans configuration files to this view, hence warning the user when attempting to delete a class that the files still refer to. JBoss, the company behind the Hibernate persistence framework [6] provides similar support in Eclipse for Hibernate Console Configuration files.

As the presence of the specialized support for all those different file types shows, the problem of external references to classes and members is a real problem in Java. RefaFlex complements this support as follows. In cases where programmers use configuration files that an IDE is not aware of, RefaFlex will successfully prevent renamings through refactorings in cases where recorded test runs use information from those files as parameters to reflective method calls. This support is generic and does not need to be adapted to the specific file format. On the other hand, RefaFlex lacks support for renaming reflective references in non-code artifacts. In those situations the fact that RefaFlex seamlessly integrates with existing refactoring support becomes important; the user gets the best of both worlds.

Sridharan et al. propose a domain specific language for specifying the (potentially reflective) behavior of web application frameworks [34], with the goal to improve static taint analysis for applications built on top of those frameworks. While these specifications considerably enrich the scope of static taint analysis, it is no option for the purpose of refactoring tools. While the approach works well with reusable frameworks, in the general case one cannot expect to have a behavior specification given for each project.

8. CONCLUSION

We have presented RefaFlex, a novel approach and tool to providing safer refactorings for reflective Java programs. RefaFlex uses runtime information from test runs to generate constraints and then considers these constraints when computing a refactoring's preconditions and transformation. It thereby guarantees that the refactoring cannot break the recorded test runs, even if they use Java's reflection API.

Our paper also highlights some interesting corner cases in which the semantics of Java's reflection API deviates from the semantics of the Java language. For example, accessibility checks in the reflection API do not at all consider the `protected` modifier. We show other interesting cases, in which behavior preservation may be unintended.

Through an empirical evaluation, we showed that our approach is correct, efficient, and can prevent transformations from being applied in all 1,358 out of 21,524 cases in which Eclipse would have incorrectly conducted the non-refactoring due to use of reflection. In 178 of those cases, RefaFlex is able to use an extended set of refactoring rules that do permit the refactoring after all, by refactoring the reflective calls themselves.

A general solution to statically resolving reflective calls does not exist. RefaFlex presents a partial but pragmatic solution to the problem. Our concepts are not restricted to Java. They could in principle be applied to other programming languages as well, given that their reflection API has a well-defined semantics.

Acknowledgements.

We wish to thank Marcus Frenkel, Mira Mezini, Andreas Sewe, Jan Sinschek, and Friedrich Steimann for their comments, which greatly helped us improve an initial draft of this paper. This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/4-1.

9. REFERENCES

[1] Java Platform, Standard Edition 6 API Specification. http://download.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html/.

[2] Joda Convert 1.1: https://github.com/jodaorg/joda-convert.

[3] Netbeans IDE: http://www.netbeans.org/.

[4] The Eclipse IDE: http://www.eclipse.org/.

[5] The Eclipse Java Development Tools: http://www.eclipse.org/jdt/.

[6] The Hibernate Persistance Framework: http://www.hibernate.org/.

[7] The Jakarta Commons Collections 4.01: http://sourceforge.net/projects/collections/files/.

[8] The Play Framework 1.1.2: http://www.playframework.org/code.

[9] The Plug-in Development Environment: http://www.eclipse.org/pde/.

[10] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, V. Russell, J.and Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[11] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[12] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, E. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM, 2006.

[13] E. Bodden, A. Sewe, J. Sinschek, and M. Mezini. Taming reflection: Static analysis in the presence of reflection and custom class loaders. Technical Report TUD-CS-2010-0066, CASED, March 2010.

[14] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, pages 241–250, 2011.

[15] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In *MSR*, MSR '11, pages 23–32, New York, NY, USA, 2011. ACM.

[16] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1:146–166, March 1989.

[17] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer, 2003.

[18] B. Foote and R. Johnson. Reflective facilities in Smalltalk-80. In *OOPSLA*, OOPSLA '89, pages 327–335, New York, NY, USA, 1989. ACM.

[19] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] R. Fuhrer, F. Tip, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, 2005.

[21] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *TOPLAS*, 29(2):11, 2007.

[22] M. Hirzel, A. Diwan, and M Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP*, pages 96–122, 2004.

[23] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Proc. 3rd Asian Symposium on Programming Languages and Systems, APLAS'05*, pages 139–160, 2005.

[24] G. McCluskey. Using Java Reflection. http://java.sun.com/developer/technicalArticles/ALT/Reflection/, 1998.

[25] A.J. Offutt and S.D. Lee. An empirical evaluation of weak mutation. *Software Engineering, IEEE Transactions on*, 20(5):337–344, may 1994.

[26] W. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[27] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[28] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3:253–263, October 1997.

[29] J. Sawin and A. Rountev. Improving static resolution of dynamic class loading in Java using dynamically gathered environment information. *International Journal of Automated Software Engineering*, 16(2):357–381, June 2009.

[30] J. Sawin and A Rountev. Assumption hierarchy for a CHA call graph construction algorithm. In *SCAM'11*, pages 35–44, 2011.

[31] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip. Correct refactoring of concurrent Java code. In *ECOOP*, pages 225–249, 2010.

[32] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *OOPSLA*, OOPSLA '08, pages 277–294, New York, NY, USA, 2008. ACM.

[33] M. Schäfer, A. Thies, F. Steimann, and F. Tip. A comprehensive approach to naming and accessibility in refactoring Java programs. *Software Engineering, IEEE Transactions on*, 99(PrePrints), 2012.

[34] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. In *OOPSLA*, OOPSLA '11, pages 1053–1068. ACM, 2011.

[35] F. Steimann, C. Kollee, and J. von Pilgrim. A refactoring constraint language and its application to Eiffel. In *ECOOP*, pages 255–280, 2011.

[36] F. Steimann and A. Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In *ECOOP*, pages 419–443, 2009.

[37] F. Tip, R. Fuhrer, A. Kieżun, M. Ernst, I. Balaban, and B. De Sutter. Refactoring using type constraints. *TOPLAS*, 33:9:1–9:47, May 2011.