From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Andreas Thies
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
andreas.thies@fernuni-hagen.de

ABSTRACT
The efficacy of mutation analysis depends heavily on its capability to mutate programs in such a way that they remain executable *and* exhibit deviating behaviour. Whereas the former requires knowledge about the syntax and static semantics of the programming language, the latter requires some least understanding of its dynamic semantics, i.e., how expressions are evaluated. We present an approach that is knowledgeable enough to generate only mutants that are both syntactically and semantically correct and likely exhibit non-equivalent behaviour. Our approach builds on our own prior work on constraint-based refactoring tools, and works by negating behaviour-preserving constraints. As a proof of concept we present an enhanced implementation of the *Access Modifier Change* operator for Java programs whose naive implementations create huge numbers of mutants that do not compile or leave behaviour unaltered. While we cannot guarantee that our generated mutants are non-equivalent, we can demonstrate a considerable reduction in the number of vain mutant generations, leading to substantial temporal savings.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *constraints.*

General Terms
Languages, Experimentation.

Keywords
Mutation Analysis, Refactoring, Testing, Accessibility, Constraints, Object-Oriented Programming.

1. INTRODUCTION
Mutation analysis (often also referred to as *mutation testing*) is the effort of detecting insufficient test coverage by making small changes to a program and observing whether these changes are caught by the present test cases [3, 7]. For a program mutation to be useful it must respect the syntactic and the semantic rules of the programming language (as checked by a compiler) while at the same time have the potential to change the observable meaning of the program (where observable here means observable by means of testing). The efficiency of mutation analysis crucially depends on the effectiveness of mutation operators in creating mutated programs — called *mutants* — that satisfy both conditions.

Refactoring is the process of changing the design of a program without changing its observable behaviour [5]. Refactoring usually involves a set of preconditions deciding whether an intended refactoring is at all possible, and a set of steps (referred to as its mechanics) prescribing the necessary program changes. While the required meaning preservation makes refactoring appear the converse of mutation analysis, both share the necessity to obey the language's syntactic and semantic rules.

Constraint-based refactoring [19–21] utilizes techniques borrowed from constraint programming to formulate and check the preconditions of a refactoring, and to ensure that performing it has the desired effect. It builds on a set of constraint rules that, when applied to a given program and its intended refactoring, produce a set of constraints whose satisfiability decides the applicability of the refactoring, and whose solution drives its mechanics.

In this paper, we show how the constraint rules of constraint-based refactorings can be adapted in such a way that they systematically generate syntactically and semantically correct mutants that likely exhibit changed behaviour. Intuitively, this can be achieved by negating precisely those constraints that are to guarantee meaning preservation in the case of refactoring — by keeping all other constraints unchanged, generated mutants are guaranteed to compile, saving the mutation process numerous vain attempts. As a proof of concept, we present experimental data obtained from applying our approach to a set of sample programs, and compare its performance to that of using a compiler for filtering out invalid mutants. Results indicate that we are not only up to two orders of magnitude faster than the compiler, but also reject large numbers of mutants that are provably equivalent.

The remainder of this paper is organized as follows. In Section 2 we motivate our work by giving an impression of the problem of generating non-equivalent program mutations and by discussing

ICSE '10, May 2-8, 2010, Cape Town, South Africa

how this has been addressed by related work. As an example and for much of the rest of this paper, we resort to mutations changing the accessibility of declared program entities, a problem identified in [12] as a prominent source of error in object-oriented programming; however, it should become clear that our approach is not restricted to precisely these kinds of mutations. In Section 3 we go into the details of our approach, and explain how it works for several different kinds of mutations. Section 4 shows how when we fail to generate mutants directly, we can still select from mutants generated by other means those that are potentially useful for mutation analysis. In Section 5, we demonstrate the effectiveness of our method by generating mutants for a number of sample programs, and comparing the results with those of a naive approach. A brief discussion of possibilities for further development concludes our work.

One more word before we start: In this paper, we refrain from formally introducing the constraints and constraint rules we used, since their sufficient explanation would have forced us to use up much of the available space. The technically interested reader is referred to [20, 21] for the origins and general working of constraint-based refactoring, and to [19] for the constraint rules relevant for the work presented here.

2. MOTIVATION

Applying mutation operators to a program without a prior specific program analysis can produce large numbers of useless mutants. Indeed, in [14] Offut reports that the fraction of useless mutants generated can grow as high as 99.97%, a negative record that has been set by his *Access Modifier Change* (AMC) mutation operator:

> *The AMC operator usually creates useless mutants. If the access is strengthened (for example, public to private), the mutated program usually does not compile — the mutant tries to use a variable that is out of scope. If the access is weakened, the mutated program is often equivalent — the mutant can still use the same variables.* [14]

At the same time, Offut recognizes that using the wrong access modifier is a relevant source of error:

> *In our experience in teaching OO software development and consulting with companies that rely on OO software, we have observed that access control is one of the most common sources of mistakes among OO programmers. The semantics of the various access levels are often poorly understood, and access for variables and methods is often not considered during design. This can lead to careless decisions being made during implementation. It is important to note that poor access definitions do not always cause faults initially, but can lead to faulty behavior when the class is integrated with other classes, modified, or inherited from.* [12]

Indeed, selecting the correct access modifiers in Java can become quite tricky. To see this, consider the following simple example in which the invocation of `m("def")` in the main method of class `B` on an instance of a class `A` is meant to bind to `A.m(String)`, but really binds to `m(Object)`:

```
class A {
  private void m(String s) { … }
  void m(Object o) { m("abc"); }
}
public class B extends A {
  void m(String s) { … }
  public static void main(String args[]) {
    A a = new A(); a.m("def");
    A b = new B(); b.m(true);
  }
}
```

To change binding, accessibility of `A.m(String)` must be increased [6, §6.6.2.1]. However, this also changes the dynamic binding of the invocation of `m("abc")` on an instance of `B` in `A.m(Object)`, which now binds to `B.m(String)` rather than `A.m(String)` as before.

Tests revealing such (potentially unintended) changes of bindings clearly have value, and so has mutation analysis showing that corresponding test cases are missing.

2.1 The Problems of Generating Mutants

Automatically creating mutants exhibiting a lack of test coverage from access modifier change requires rather intimate knowledge of the Java language specification [6]. For example, Java syntax mandates that not all access modifiers can be used in all places: reducing accessibility of a method declared in an interface to a level below *public* (the default) causes a compile error, since all interface methods must be publicly accessible; changing accessibility of a top-level class to *protected* causes an error, since *protected* is not allowed for top-level classes; and so forth.

While such vain applications of the AMC operator can be prevented by obeying rules of Java syntax, others require a deeper, semantic analysis. For instance, while increasing accessibility of method `m()` in class `A` from *default* (no access modifier given) to *protected* or *public* presents no problem, reducing it to *private* will lead to a compile error, since `m()` is no longer accessible from `B` (the example assumes that `A` and `B` reside in the same package):

```
class A {
  void m(){ … }
}
class B {
  void n(){
    (new A()).m();
  }
}
```

Preventing vain applications of this kind is more difficult than in the examples based on syntax only, since it requires an analysis of the program with regard to references to the declared entity whose accessibility is to be changed. However, while this task may be time-consuming, it is not technically demanding, and therefore could also easily be incorporated in a mutation analysis tool that has AMC in its repertoire.

As it turns out, however, checking for existing references does not suffice to ensure the validity of an access modifier change. For example, the subtyping rules of Java mandate that the declared accessibility of an overriding method must not drop below that of the method it overrides [6, §8.4.8.3]. Therefore, if the above code example is extended by

```
class C extends A {
  void m(){ … }
}
```

in the same package, raising accessibility of method `m()` in `A` above *default* would lead to a compile error.

While this problem is still fairly obvious (and easy to check), there are others that are far more obscure, particularly if they involve chains of relationships. For instance, given the Java code

```
interface I {
  void m();
}
class A {
  public void m() { }
}
class B extends A implements I {}
```

reducing accessibility of m() in A to *protected* would lead to a compile error because A.m() contributes to implementing the interface I in A's subclass B and therefore has to be declared *public*. Although all problems of this kind will eventually be caught by the compiler, (ab)using the compiler as an oracle deciding whether a generated mutant presents a valid program and is thus acceptable, is rather expensive, especially if the rejection rate is high (Section 5 will present concrete numbers corroborating this).

However, generating *valid mutants*, i.e., mutants that represent programs that are free from syntactic and semantic errors[1], is only the first hurdle in mutation analysis; the second, and at least as demanding, is to generate mutants that actually change the meaning of a program. To see why this is a problem (again for the case of changing accessibility), the following example is instructive:

```
public class A {
  void m() { … }
  public static void main(String args[]) {
    A a = new B();
    a.m();
  }
}
public class B extends A {
  public void m() { … }
}
```

If A and B reside in the same package, increasing the accessibility of m in A to *protected* or *public* is valid, but useless for the purpose of mutation analysis, since it does not change the meaning of the program (the invocation of m() on a in A is bound to B.m() before and after the mutation). If however A and B reside in different packages, an increase of accessibility changes binding (the call of m() on a in A is bound to A.m() before and to B.m() after the mutation), potentially changing the meaning of the program. Whether the latter is actually the case depends on whether the implementations of m() in A and B have different effects, which escapes analysis in the general case (and in any case is not our topic here); however, since a change of binding is a change of program the programmer should at least be aware of, we call such mutants *relevant*. For the remainder of this paper, the goal is to *compute relevant mutants*; we leave it to further inspection to decide whether a relevant mutant is equivalent to the original program or calls for additional test cases.

Note that, while the AMC operator may seem somewhat exotic to focus on, the problems it reveals are representative of a much broader class of errors in object-oriented programming, namely the wrong binding of references in the context of overloading and overriding (and, through a trick, also hiding). Changing access modifiers is a particularly simple way to provoke such errors, and would therefore be an effective means to generate mutants revealing a lack of test coverage — if it were not for the high number of irrelevant mutants commonly produced.

[1] We use the term "semantic error" here to denote errors reported by the semantic analysis performed by the compiler (i.e., typing errors etc.). They are not to be confused with logical errors.

2.2 Related Work

Budd and Angluin showed that even in the limited domain of testing, it is in generally undecidable whether a mutated program generates the same output as its original [2]. Despite this fundamental insight, and because non-equivalence is crucial for mutation analysis, several attempts have been undertaken to reduce the human effort of eliminating generated mutants that are useless.

Early attempts go back to Baldwin and Sayward [1], who have used compiler optimization techniques to filter out equivalent mutants. The main idea behind this is that code optimization tasks done by the compiler (as prescribed by code optimization rules) effectively produce equivalent mutants. Vice versa, an equivalent mutant can be described as a compiler optimisation: if a mutant falls under a compiler optimisation rule (either as input or as output), it is equivalent. However, a later implementation of this approach [13] showed only low success rates: in an empirical analysis, only 10 % of equivalent mutants could be detected.

Other advanced approaches use control flow graphs to detect equivalent mutants [16]. Especially, knowledge of infeasible paths can help to reduce the total amount of equivalent mutants because mutating an unreachable statement will not change the semantics of the program. In this context, constraint-based approaches were first mentioned [16]. Firstly, the reachability of a node depending on the input in the original program can be formulated with constraints. Secondly, constraints can be generated that judge whether a certain mutant will change the state of the program after execution of the mutated statement [16]. This latter technique has also been used to generate test data automatically [4]. In subsequent work, an implementation was given that detects equivalent mutants using these principles [15], and it has been shown in an accompanying empirical study that this approach leads to a detection rate of almost 50 % of all equivalent mutants. The main disadvantage of this approach is that it only considers the state of the program immediately after execution of the mutated statement (where it may remain unobservable). In particular, it cannot tell whether the change of state is propagated to the program's output (so-called weak mutation testing [10]).

To tackle this latter problem program slicing has been suggested [9]. The basic idea here is to reduce the size of problem by slicing techniques so that only the relevant part of code on which a mutant might have impact has to be analyzed. Basically, this helps the programmer to decide faster and more reliably if a so-called *stubborn mutant* [8] (i.e., a mutant that changes the state of the program but has no impact on the program's output) not detected by the constraint-based approach is equivalent or not.

Another technique enhancing the constraint-based approach of [15] is based on program dependence analysis [8]. For this, a tool is offered to assist the programmer in detecting equivalent mutants by pointing out certain variables of interest. Also, it is shown how a dependence analysis can help to avoid a special kind of equivalent mutants before the constraint analysis.

A recent approach to filtering out equivalent mutants relies not on checking constraints, but on checking program invariants: The JAVALANCE-Tool [18] first learns invariants from runs of non-mutated programs and then checks for violations of these invariants by mutated programs. An empirical evaluation conducted on several open source programs showed that mutants violating multiple invariants are likely non-equivalent.

All techniques mentioned above have in common that they target imperative programming constructs without taking object oriented principles such as dynamic method binding and encapsulation into account. The results of first experiments on equivalence rates of mutation operators designed specifically for object-oriented programs can be extracted from [12]. In this work, a large number of different mutation operators for Java have been proposed that account for several object-oriented principles, and it was assumed that some of these operators will generate significantly more equivalent mutants than others. Empirical evidence corroborating these assumptions has been given in a subsequent study [14], in which the equivalence rates ranged from 99.97 % for the *Access Modifier Change* operator to 0.22 % for an operator which deletes explicit calls of a parent's constructor. It remains unclear from this work, however, how equivalence was decided, in particular, how much manual effort this involved.

3. CONSTRAINT-BASED PROGRAM MUTATION

As mentioned in the introduction, program mutation shares with refactoring the condition that executability of programs must be maintained, and differs from it by its need to change behaviour. Leaving syntactic and semantic errors aside, what is a failure for refactoring is thus a success for program mutation. To explain how we exploit this relationship systematically, we first take a brief look at refactoring, specifically constraint-based refactoring.

3.1 Constraint-based Refactoring

In constraint-based refactoring, a set of constraint generation rules is applied to a program and its intended refactoring. The constraint rules represent the syntactic and semantic rules of the programming language and the generated constraints represent these rules as applied to those elements of the program that are involved in the intended refactoring. For instance, a basic rule of Java requires that if a name, *r*, references a declared entity, *d*, the entity's declared accessibility, $\langle d \rangle$, must be at least *private* if *r* resides in the same class, at least *default* if it resides in a different class, but same package, and so forth. This is expressed as a constraint

$$\langle d \rangle \geq \alpha(\lambda(r), \lambda(d))$$

in which λ is a function mapping program elements to their locations in the program (usually the body of a package or type) and α is a function computing the minimum required accessibility of the second location to be accessible from the first. Note that in Java, access modifiers are totally ordered, but constraints can also be formulated for unordered access modifier sets, and even for languages without access modifiers (such as Eiffel) [19].

By putting them into relation with each other or with constant values, each generated constraint constrains one or more constraint variables. Constraint variables represent changeable properties of program elements; these are typically the types of declared entities, the locations where they are declared, or their accessibility ($\lambda(r)$, $\lambda(d)$, and $\langle d \rangle$ in the above example).

All constraint variables have initial values, which are derived from the program as is; in the previous example, the values correspond to the actual declared accessibility of *d* and the actual locations of *r* and *d*. Taking this initial variable assignment, the constraint system (set of generated constraints, consisting of a single constraint in the above example) is always solved (or the constraint rules are inconsistent with the language specification, or the program is syntactically or semantically incorrect, i.e., does not compile).

The principle behind constraint-based refactoring is that a refactoring may assign the variables of the generated constraints new values (representing the changes of the refactoring) as long as the constraint system remains solved; if it does not, the refactoring is illegal and cannot be performed. A refactoring intending to change one or more variable values (for instance by changing type, accessibility, or location of one or more declared entities) therefore has to check whether this invalidates the generated constraint system; if so, the refactoring must either be refused or search for new values for the constraint system's other variables that make it solved again. The new values then represent additional changes to be performed by the refactoring.

If all generated constraints must be satisfied for a refactored program to maintain its executability (i.e., freeness of compile-time errors) and behaviour, it follows that if a constraint is violated, the program has lost at least one of these properties. Deliberately violating certain constraints by assigning constraint variables suitable values can thus lead directly to relevant mutants. The question is which constraints to violate.

3.2 Generating Relevant Mutants from Accessibility Constraints

In previous work of ours on improving accessibility-related refactorings [19], we have identified a set of 16 constraint rules that capture access control of Java. Of the 16 rules, 11 prevent changes of accessibility that would lead to a compile error; transferred to the problem of mutant generation, they serve to prevent invalid mutants and therefore must be left untouched. Of the remainder, four rules prevent changes of binding that, unless covered by corresponding test cases, will go unnoticed; in our current setting, they would prevent changes of access modifiers representing relevant mutants and thus what we are interested in. Changing these constraint rules by negating the generated constraints would invert their effect, i.e., generate constraints that are satisfied if and only if the binding changes; they would lead us directly to relevant mutants. The last constraint rule prevents changes of accessibility that, depending on the actual program, either lead to a compile error or to a change of binding; it will be treated separately in Section 3.2.3.

3.2.1 Loss of Dynamic Binding

The first constraint rule whose negation[2] leads to relevant mutants is one that prevents a loss of dynamic binding. Basically, it states that if a method *d′* is to override a method *d*, the declared accessibility of *d* must be at least the accessibility that is required for an access of *d* from the location of *d′*. For the sample program

```
class A {
  void m() { … }
  void n() { m(); }
}
class B extends A {
  void m() { … }
}
```

[2] In the following, when we speak of negating a constraint rule, we mean that the rule is changed to produce a constraint that is the negation of the constraint produced by the original rule.

the generated constraint is

$$\langle \texttt{A.m()} \rangle \geq \alpha(\lambda(\texttt{B.m()}), \lambda(\texttt{A.m()}))$$

in which α evaluates to *default* if both classes reside in the same package. Negating this constraint (i.e., replacing ≥ with <) suggests either lowering the accessibility of A.m() to *private* or moving B to another package (so that α evaluates to *protected*), both with the effect that a call of m() in n() on instances of B is no longer dispatched to B.m(). Other constraint rules (which are not negated) take care that neither change leads to an invalid mutant (i.e., to a mutant that does not compile); for instance, if A.m() is referenced from outside of A, a corresponding constraint would prevent lowering the accessibility of A.m() to *private* as a possible mutation.

3.2.2 Introduction of Dynamic Binding

The converse of losing dynamic binding is introducing it where it was previously absent, which can also change the meaning of a program. The corresponding constraint rule from [19] states that if a method *d'* in a subclass exists whose signature is a subsignature [6, §8.4.2] of a method *d* in a superclass, but *d'* does not override *d*, declared accessibility of *d* must not be increased to values allowing it to be accessed from the location of *d'* (since then *d'* would override *d*). For the program

```
class A {
  private void m() { … }
  void n() { m(); }
}
class B extends A {
  void m() { … }
}
```

(again both classes in same package), the generated constraint is

$$\langle \texttt{A.m()} \rangle < \alpha(\lambda(\texttt{B.m()}), \lambda(\texttt{A.m()})) \quad (= \textit{default})$$

If the negated rule is applied to the program every solution to the generated constraint set would have to increase the declared accessibility of A.m() to values above *private*, thus introducing the dynamic binding of m() for instances of B. Again, that such a mutation does not invalidate the program for other reasons is checked by other constraints.

3.2.3 Static Re-binding in Presence of Overloading

In Java, a change of binding due to a change of accessibility is not limited to the dynamic case. For instance, in the program

```
class A {
  void m(Object o) {…}
  private void m(String s) {…}
}
class B {
  void n() { (new A()).m("abc"); }
}
```

raising the accessibility of A.m(String) above *private* will re-bind the call of m(String) from C, potentially changing the meaning of the program. For the refactoring case, this is prevented by a constraint rule saying that accessibility of an overloading method that is more specific (in terms of its formal parameter types; here: A.m(String)) than the one a method call currently binds to (here: A.m(Object)) must remain on a level that leaves it inaccessible for the location of the method call:

$$\langle \texttt{A.m(String)} \rangle < \alpha(\lambda(\texttt{B.n()}), \lambda(\texttt{A.m(Object)}))$$

Again, negating this constraint means demanding a change of accessibility and therefore leads to a change of binding, potentially changing the meaning of the program. Thus, by replacing the original constraint rule with its negation, a constraint solver can generate relevant mutants directly.

As in the dynamic binding cases above, re-binding to an overloading method due to its increased accessibility has a converse that results from restricting accessibility. This follows directly from the previous example, when the increased accessibility of A.m(String) is set back to *private*. However, in the refactoring setting we did not prevent this change of binding by a special constraint rule: there, the general accessing rule (used as an example in Section 3.1) requiring that a called method remains accessible from the call site suffices. For the purpose of mutation analysis, however, for which a change of binding is to be forced if possible, two cases must be differentiated: whether a call binds to a method for which a less specific overloading exists to which the call could bind alternatively, or whether no such alternative binding possibility exists. In the first case, the constraint demanding accessibility must be inverted (to force a change of binding), while in the second it must be maintained as is (to prevent a compile error). This case analysis means that if the same set of constraint rules is to be used for both purposes (refactoring and mutation analysis), the accessibility constraint rule has to be split in two, one with the additional antecedent that an overloading suitable for a re-binding is present, the other with the same additional antecedent negated. Note that in the refactoring case, this split has no effect (the antecedents complement each other and the consequent is the same). In the mutation analysis case, the former rule must be negated, while the latter must remain as is.

3.3 Generating Mutants by Introducing or Deleting Entities

Somewhat related to a change of static binding due to overloading is the change of binding caused by hiding [6, §8.4.8.2].[3] It is different, however, in that hiding does not depend on the choice of access modifiers: it is necessary and sufficient for hiding that the hiding element is present (and for a lack of hiding that it is absent).

For the refactoring case, we were able to treat hiding under the umbrella of accessibility by introducing a fifth, smallest access modifier, which we call *absent* [19]. It represents complete inaccessibility of a declared program element. Being able to assign this value to a constraint variable representing accessibility without violating a constraint means that the corresponding declared element can be deleted without altering the compileability or behaviour of the program; a constraint requiring that a variable has *absent* as its value either means that the corresponding element must be deleted (if it is present) or that it must not be introduced (if it is not yet present). However, as we will see next, unlike for refactoring, constraints of the latter kind cannot be generated for mutation analysis.

[3] Similarity extends to the degree that like overloading, hiding requires a splitting of the general accessing rule (which is sufficient for the refactoring case) into one that is required to prevent compile errors and one to prevent a change of binding.

3.3.1 The Difficulty of Introducing Elements with Constraints

In order to prevent hiding of a declared element (field or static method) in the course of a refactoring, a constraint must be generated that requires the hiding element's accessibility to have the value *absent*. Negating such a constraint would mandate introducing a declared entity that hides another, which would indeed be a relevant mutation operator (for the case of fields called *Hiding Variable Insertion* in [12]). However, since the program entity is absent in the original program, there is no constraint variable associated with it whose value could be set to an accessibility level above *absent*.

In the case of refactoring, we have solved this problem by not applying the constraint rules to the program as is, but to the program with the intended refactoring applied, which may try to introduce the declared entity in a location (or rather move an existing one to a location) in which it is not allowed. A constraint generated from the changed program saying that this entity should be absent then leads to a conflict that cannot be resolved, so that the refactoring is refused. Because this requires foresight of the refactoring, we have called these applications of constraint rules *foresight applications* [19].

Unfortunately, foresight applications of rules are not suitable for the purpose of generating mutants, since the required foresight would need to know the program change the (negated) rule's application is to provoke. However, as we will see in Section 4, foresight applications of rules can nevertheless contribute to generating relevant mutants, by rejecting the irrelevant ones from a set of mutants generated by other means.

3.3.2 Deleting Entities with Constraints

The converse of introducing declared entities is deleting them. Deletion of an entity is accepted by the compiler if the references to it can be re-bound to another entity, for instance one that was hidden by the deleted entity. Deleting entities can therefore also generate relevant mutants; the corresponding mutation operators have been called *Overloading Method Deletion*, *Hiding Variable Deletion*, and *Overriding Method Deletion* in [12].

Deleting a declared entity can be seen as a stronger form of changing its accessibility to a value so low that it becomes inaccessible. Indeed, there are situations in which accessibility of an entity cannot be reduced further, but in which it can be deleted: for instance, in

```
class A { int i = 1; }
class B extends A {
  private int i = 2;
  void m() { if (i == 2) … }
}
```

deletion of `i` in `B` is possible (but leads to a change of binding).

Exploiting our (virtual) access modifier *absent*, our accessibility constraint rules from the previous sections can also be used to generate relevant mutants that result from deleting declared entities. This is possible by making the constraint solver consider *absent* as a possible value for the declared accessibility of an element in question. However, including *absent* as a possible, smallest value of accessibility requires adaptation of certain constraint rules: for instance, while subtyping mandates that the accessibility of an overriding method must not drop below that of the method it overrides, *absent* (which is lower than *private*) may be a possible value (if other generated constraints permit).

3.4 Generating Relevant Mutants from Type Constraints

Our approach to generating relevant mutants from negated refactoring constraints works not only for accessibility — it can also be transferred to other types of constraints. For instance, the type generalization refactorings GENERALIZE DECLARED TYPE and USE SUPERTYPE WHERE POSSIBLE [20] change the values of constraint variables representing the types of declared entities from their current types to supertypes (if possible). Similar to changing accessibility, changing the type can lead to a change of binding, as the following example demonstrates:

```
public class A {
  A a;
  void m(Object o) { … }
  void m(B b) { b.m(a); }
}
public class B extends A {}
```

Here, the type of the formal parameter `b` in method `A.m(B)`, `B`, must not be changed to `A`, but not because `A` is not a sufficient abstraction (generalization) of `B` for the needs of `b` (only `m(Object)` is required of `b`, and this is defined in `A`), but because changing the signature of `A.m(B)` to `A.m(A)` leads to a re-binding of the call `b.m(a)` in `m(B)` (from `m(Object)` to `m(A)`), thus changing the meaning of the program. Negating the constraint rule that prevents the generalization in such cases directly produces relevant mutants of this kind.

3.5 Completeness of Constraint-based Mutant Generation

One question that remains is whether our approach is complete in that it generates all relevant (and therefore also all non-equivalent) mutants resulting from access modifier changes. Provided that our constraint solver generates all solutions for a generated constraint system, this question boils down to the question whether our constraint rules completely and correctly model the language specification with regard to accessibility. While proofs of this kind are generally difficult to provide, we maintain that our accessibility constraint rules, which are generally application-independent, have been tested extensively in the context of refactoring [19], and tested further in this present work, by checking that running the regression tests for compiling mutants that have been rejected as irrelevant indicate no change of behaviour.

4. CONSTRAINT-BASED MUTANT REJECTION

Constraints cannot only be used to systematically generate relevant mutants, they can also be used to reject as irrelevant mutants generated by other means. This capability is of particular interest to us when relevant mutants cannot be generated by negating constraint rules, as was described in Section 3.3.1.

To get an impression of how this works, assume that we have a mutant generator that uses some heuristics to make changes to a program by inserting new program elements. Essentially, if these heuristics do not capture at least the knowledge about the target language that is captured in refactoring constraints, it is bound to

Table 1. Sample projects used for the evaluation

Project	*Number of classes*	*Number of test cases*	*Source*
JUnit 3.8.2	102	102	junit.org
JHotDraw 6.01b	405	1160	www.jhotdraw.org
Draw2D 3.4.2	347	173	www.eclipse.org/gef
Jaxen 1.1.1	213	2030	jaxen.org
HTMLParser 1.6	165	669	htmlparser.sourceforge.net

produce mutants that are invalid or irrelevant. For instance, given the program (all types in same package)

```
class A {}
interface I { static int i = 2; }
class B extends A implements I {
  void m() { int j = i; }
}
```

such a mutant generator might insert the declaration `static int i` into `A`, which would lead to a compile error (access to `i` in `B` becomes ambiguous). As it turns out, such a change of program is rejected by a corresponding constraint rule defined in [19]; applying this constraint with foresight of the introduction leads to the constraint

$$\langle \mathsf{A.i} \rangle < \alpha(\lambda(\mathsf{B.m}()), \lambda(\mathsf{A.i})) \quad (= \textit{default})$$

which is not satisfied for $\langle \mathsf{A.i} \rangle = \textit{default}$ and thus signals an invalid mutant.

Now a mutant generator respecting the syntactic and semantic rules of Java (i.e., restricting itself to generating mutants that compile) applied to the above example might still insert `static int i` into `B`. How, then, can be known that such an insertion produces a mutant relevant for mutation analysis? Again, evaluating the constraints generated for refactoring of accessibility will indicate that a corresponding refactoring could not be performed, but this time the failing constraint

$$\langle \mathsf{B.i} \rangle = \textit{absent}$$

is one resulting from foresight application of the rule preventing hiding that was described in Section 3.3.1 as non-negatable; because it is a meaning preserving rule, its violation identifies a relevant mutant.

Note that even without resorting to foresight rules, constraints can be used for rejecting irrelevant mutants. The idea here is that if evaluating the constraints generated after a mutation suggest that undoing the mutating change (for instance by removing an inserted entity, by setting its accessibility to *absent*) potentially changes the meaning of the program and therefore would have to be rejected in the refactoring case, then it follows that the mutation potentially changed the meaning in the first place, and therefore produced a relevant mutant. Note that this approach works without the negation of constraint rules; however, it requires a new constraint generation after each mutation and is therefore more expensive (cf. its evaluation below).

5. EVALUATION

To get an impression of the effectiveness of our approach to constraint-based mutant generation, we have applied it to several open source programs. The chosen programs are listed in Table 1;

Table 2. Results for naive changes of access modifiers

Project	*Number of mutations*	*Mutants*			
		invalid		*valid*	
		synt.	*sem.*	*irrelev.*	*relevant*
JUnit	3063	1089	625	1346	3
JHotDraw	14977	5841	3548	5536	52
Draw2D	14280	2421	4453	7374	32
Jaxen	6159	1562	2689	1903	5
HTMLParser	8478	1993	2714	3754	17
total	46957	12906	14029	19913	109
	100%	27%	30%	42%	0,23%

the accompanying test suites were only used to test our approach, i.e., to check whether any of the mutants classified as irrelevant caused a failing test case (which would have been indicative of a false classification; cf. the discussion in Section 3.5).

Table 2 shows the filtering performance of all constraint rules governing the change of access modifiers (including those that keep a program compilable and those that keep its binding unchanged, the latter negated) when jointly applied to the projects from Table 1. The number of mutations is the number of access modifier changes that is at all possible (counted naively, i.e., without considering even the simplest syntactic rules such as that interface methods must be public). The columns labelled "invalid" counts those mutations that must be filtered out because the resulting programs will not compile, for either syntactic or semantic reasons (cf. Section 2.1). The columns labelled "valid" count those mutants that will compile, differentiating between those in which the binding remains unchanged (which is why they are irrelevant to mutation analysis) and those with changed binding. It is this last column that reveals the value of our approach.

As can be seen from Table 2, the syntactic constraints (which any reasonable mutation operator would be expected to respect) are only sufficient for filtering out one quarter of all mutants; the remaining three quarters must be subjected to deeper analysis. Constraints covering the (static) semantics (accessing, subtyping, etc.) eliminate another 30% of all mutants; achieving this effect requires a whole-program analysis that most mutant generators would leave to the compiler (which will reject invalid mutants, albeit only at a high price: see Table 3 for the time required by the compiler as compared to our constraint approach). The compiler will however not be able to filter out those mutants that are syntactically and semantically correct ("valid"), but do not lead to a change of binding and thus meaning ("irrelevant"): these cases, which make up for more than two fifth of all mutations, present the greatest challenge for mutant generators — and yet, using our approach they are classified on the same basis as all others. The remaining less than 1% of all possible mutants are the only ones worth looking into — all other mutants are generated in vain. The savings made possible by our approach are thus considerable.

Table 3 shows the performance of our approach when compared to elimination of invalid mutants as performed by the compiler (all times measured on a 2.1 GHz Intel dual core processor with 2 GB of main memory, running Windows XP and the Eclipse 3.4 compiler in incremental mode). To be fair, we fed only syntactically valid mutants into the evaluation (as argued above, their fil-

Table 3. Compiler vs. Constraints (times in secs)

Project	*Syntactically valid mutants*	*Of these semantically invalid as determined by the compiler*			*Semantically invalid or irrelevant as determined by the constraints*		
		abs.	*rel.*	*time*	*abs.*	*rel.*	*time*
JUnit	1974	1714	87%	194	1971	99.8%	1.9
JHotDraw	9135	5841	64%	1330	9084	99.4%	30.8
Draw2D	11859	4453	38%	1431	11827	99.7%	17.8
Jaxen	4597	1903	41%	307	4592	99.9%	3.6
HTMLParser	6485	3754	58%	739	6468	99.7%	7.8

tering is so simple that it can easily be performed by even the most naive mutant generators). On top of that, we did not restrict evaluation of the constraints to those equivalent to the semantic rules checked by the compiler, but included them all, resulting in a much higher rejection rate. As can be seen, even under these unequal conditions, the constraint-based approach is between 43 and 102 times faster than using the compiler. This is substantial.

Since we must assume that not all constraint rules contribute to the results equally [11, 19], we have also evaluated them separately. Table 4 shows the results of negating the constraint rules preventing a loss of dynamic binding (Section 3.2.1), preventing its accidental introduction (Section 3.2.2), and preventing a change of static binding in presence of overloading (Section 3.2.3; no cases in which increasing accessibility would have changed binding were found). The column headed "opportunities" counts the number of applications of the negated constraint rule and is thus a measure of the number of possibilities for generating relevant mutants. The column headed "relevant mutants" counts the number of cases in which the binding could actually be changed by altering an access modifier. In all other cases, the alteration violated syntax or semantics-preserving constraints and was thus refused. "Time" measures the time needed to generate the mutants; it includes generation of the constraints (including building the AST), their negation, and solution of the constraint system.

As correctly predicted by Offut [15], the number of relevant mutants produced, although rather unevenly distributed among the different constraint rules, is generally small. Yet, one must bear in mind that using our approach, the number of irrelevant mutants potentially generated by other approaches is of no interest, and that the produced relevant mutants are indeed representative of common (and difficult to detect) programming errors. Therefore, Offut's conclusion, that implementing the AMC operator is not worthwhile (because it "generates uncompilable, equivalent or redundant mutants, and is [therefore] not needed in MuJava" [15]), seems somewhat premature.

Table 5. Mutants resulting from deleting an overloaded method or a hiding entity

Project	*Opportunities*		*Relevant mutants*	
	overl.	*hiding*	*absolute*	*relative*
JUnit	2	0	1	50%
JHotDraw	9	1	1	11%
Draw2D	5	2	5	71%
Jaxen	0	0	0	
HTMLParser	0	0	0	
total	16	3	7	37%

Table 5 shows the number of mutants that were generated by deleting overloading methods (the OMD operator [14]) or a hiding entity (field or static method; cf. Section 3.3.2). Its results for overloading include those of Table 4, since in all cases in which a reduction of accessibility was possible, the method could also be deleted. Note that by introducing the access modifier *absent*, and viewing deletion as an extreme case of access restriction, the redundancy in the results of the two mutation operators can be avoided, simply by merging them into one. That this must not lead to rejection of the AMC operator as suggested in [15] follows from the other results of Table 4, which cannot be subsumed by OMD.

Last but not least, we have measured the performance of our approach for constraint-based mutant rejection as described in Section 4, for the example of inserting hiding variables (the IHI mutation operator [12]). For this, we have implemented a simple mutant generator that inserts variable declarations in subclasses that also exist in superclasses, and that makes sure that the generated mutants are valid, i.e., still compile. Depending on the references existing in the program, each such mutation may, but need not, introduce hiding. We then check whether the obtained mutants are relevant by checking whether our constraints allow the mutations to be undone (the inserted variable to be removed); if so, the mu-

Table 4. Mutants dropping overriding, introducing overriding, and making an overloaded method inaccessible

Project	*Dropping overriding*				*Introducing overriding*				*Making an overloaded method inaccessible*			
	Opportunities	*Relevant mutants*		*Time (secs)*	*Opportunities*	*Relevant mutants*		*Time (secs)*	*Opportunities*	*Relevant mutants*		*Time (secs)*
		abs.	*rel.*			*abs.*	*rel.*			*abs.*	*rel.*	
JUnit	156	3	2%	2.2	0	0		2.5	2	0	0%	1.1
JHotDraw	2017	48	2%	27.4	4	3	75%	8.6	9	1	11%	1.3
Draw2D	1205	29	2%	18	0	0		6.4	5	3	60%	7.7
Jaxen	785	5	6%	3.6	0	0		3.0	0	0		2.5
HTMLParser	720	17	2%	7.6	0	0		3.7	0	0		3.8
total	4883	102	2%	58.8	4	3	75%	24.2	16	4	25%	16.4

Table 6. Constraint-based rejection of irrelevant mutants inserting hiding variables, generated by other means

Project	*Valid mutants submitted*	*Rejected as irrelevant*			*Running time of test suites (secs)*	
		abs.	*rel.*	*time (secs)*	*all subm.*	*relev. only*
JUnit	67	67	100%	68	51	0
JHotDraw	815	801	98%	5179	2967	51
Draw2D	603	524	87%	4061	3534	463
Jaxen	87	82	94%	225	356	20
HTMLParser	182	165	91%	779	2844	266

tant is irrelevant (does not introduce hiding), if not, it is indeed relevant. The results of this procedure are shown in Table 6.

As can be seen, the constraints reject as irrelevant large numbers of valid mutants inserting variables and are therefore again highly effective. However, the times required for rejection are much longer than for generation (cf. Tables 3–5): this is so because for rejection, the constraint system has to be built anew for every mutant (so that it can be checked whether undoing the mutation itself makes a relevant mutation), whereas for generation, the constraint system need be built only once, since for computing the different mutants, only existing constraints are negated. And yet, when comparing this time with the time saved by running test suites on fewer mutants (for mutants proven to be irrelevant, running the tests is useless: unless our constraints are flawed, there will be no failures), the time required for constraint generation is well spent: on average across all five sample projects, rejection takes only slightly longer than running the test cases for irrelevant mutants. In practice, there will be a huge saving, however: finding out that a lack of failure of a test suite is not due to a lack of coverage, but to the irrelevance of the mutant, will take indeterminately longer. This is where the true saving of our approach comes from.

6. FURTHER DEVELOPMENT

6.1 Mutant Rejection Based on Other Refactoring Techniques

The idea of rejecting irrelevant mutants generated by other means (Section 4) can also be transferred to the reasoning behind the meaning-preserving mechanics of refactorings that are not constraint-based. For instance, in [17] Schäfer et al. developed the mechanics for the RENAME refactoring necessary to maintain binding of names to their originally referenced declared entities that are hidden (by nesting or by inheritance) after the renaming, by introducing sufficient qualification of names. Thus, when a mutant is generated by renaming a program entity, it can be checked whether this leads to a change of binding (this is the case when the corresponding refactoring decides that a qualification is necessary); in the positive case, a relevant (potentially useful) mutant has been found; in the negative case, it is useless and can be rejected.

6.2 Enhancing Mutant Reports

A conventional mutant generation tool would know what it changed, but not necessarily why, or where, this change led to a change of meaning of the program. For instance, if changing the accessibility of a declared entity changes the binding of a reference, perhaps even one originally referring to another entity, this is not at all obvious from the changed location. For instance, if the mutant consists of changing accessibility of `m(String)` in

```
package a;
public class A {
  int m(String s) { … }
}
```

to *public*, it is not at all clear why, or where, this imposes a change of meaning. Only by looking at the whole program, which might include

```
package b;
public class B extends a.A {
  public int m(Object o) { … }
}
```

```
package a;
public class C {
  void n() {
    int i = (new b.B()).m("abc");
  }
}
```

the problem becomes clear: the (static) binding of the call of `m` in `C` is changed to `B.m(Object)`. Generally, the manifestations of a change of meaning can be spread throughout the whole program and may be nontrivial to detect. The manifestations are however as interesting as the mutant itself, in particular if the goal is to create new test cases (see below).

By negating a constraint that was to prevent a change of meaning, we know up-front where the meaning is changed, namely the place in the source code from which the constraint was generated. In the previous example, the constraint generated from the reference to `B.m(Object)` in class `C` is that there must be no other (overloaded) version of `m` with more specific parameter type that the reference can bind to, specifically that `A.m(String)` must remain inaccessible:

$$\langle \texttt{A.m(String)} \rangle < \alpha(\lambda(\texttt{C.n()}), \lambda(\texttt{B.m(Object)})) \quad (= \textit{public})$$

Violation of this constraint guarantees us that the reference will bind differently, which is the manifestation of the (potential) change of meaning. The other information of interest, which change caused the change of meaning (the increasing of the access modifier of `A.m(String)`), is computed in the course of the solution of the invalidated constraint system and can — as derived information — also be displayed.

6.3 Deriving Missing Test Cases

The ultimate goal of mutation analysis is to generate missing test cases. From the mutated source location alone, however, such missing test cases may be hard, if not impossible, to derive: continuing the example from the previous subsection, from inspecting the changed class, `A`, and its test cases, it is not at all clear if, and if so which, test cases are missing. Certainly, one could write a test case failing if accessibility is changed to what is suggested by the mutant, but this test case would be meaningless in terms of documentation, since it is unclear why it is there, and therefore also when a change of program design would allow dropping it.

By systematically exploiting the information that is in knowing which constraint was negated, and which program elements it was generated from, a test case stub can be automatically generated that points to the heart of the problem:

```
@Test
public void testMString() {
  //TODO Auto-generated test object
  B b = null;
  //TODO Auto-generated test parameter
  String param = null;
  //TODO Auto-generated test expectation
  Integer expected = null;
  Assert.assertEquals(expected, b.m(param));
}
```

Generally, however, our approach will not be able to provide the necessary set up (fixture) of the test case, nor provide a suitable test oracle.

6.4 Porting to Other Languages

Another inherent advantage of our approach is that it is easily ported to other languages. As pointed out in [19], constraint-based refactoring tools can easily be transferred to other programming languages, simply by replacing the constraint rules representing the language's syntax and semantics. Given the close connection between refactoring and mutation analysis which our approach exploits, the constraint rules for the different languages need to be generated only once, and can be used without further effort in both applications.

7. CONCLUSION

We have argued that the knowledge about a programming language's syntax and semantics as captured in refactoring tools, constrained-based ones in particular, can be reused to systematically mutate programs in such a way that the resultant mutants are guaranteed to be executable, while at the same time likely exhibit changed behaviour. Based on a set of sample programs and their accompanying test suites, we have demonstrated that our approach is capable of producing mutants likely exhibiting changed behaviour efficiently, avoiding the large numbers of vain mutant generations other approaches are suffering from. Our results are the more pleasing as reaching them is almost free: for most cases, all that needs to be done is to negate the semantics-preserving rules of existing formalized refactoring tools.

8. REFERENCES

[1] Baldwin, D., and Sayward, F. 1979. Heuristics for determining equivalence of program mutations. Technical Report 161. Yale University, Dept. of Computer Science.

[2] Budd, T., and Angluin, D. 1982. Two notions of correctness and their relation to testing. Acta Informatica. 18, 1 (November 1982), 31–45.

[3] DeMillo, R., Lipton, R., and Sayward, F. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. Computer 11, 4 (April 1978), 34–41.

[4] DeMillo, R., and Offutt, J. 1991. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17, 9 (September 1991), 900–910.

[5] Fowler, M. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley 1999.

[6] Gosling, J., Joy, B., and Steele, G. 2005. The Java Language Specification, Third Edition. Addison-Wesley. http://java.sun.com/docs/books/jls/.

[7] Hamlet, R. 1977. Testing Programs with the Aid of a Compiler. IEEE Transactions on Software Engineering 3, 4 (July 1977), 279–290.

[8] Harman, M., Danicic, S., and Hierons, R. 2000. The Relationship Between Program Dependence and Mutation Analysis. In Proceedings of Mutation 2000: Mutation Testing for the New Century (2000), 5–13.

[9] Hierons, R., Harman, M., and Danicic, S. 1999. Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability 9, 4 (December 1999), 233–262.

[10] Howden, W. E. 1982. Weak Mutation Testing and Completeness of Test Sets. IEEE Transactions on Software Engineering 8, 4 (July 1982), 371–379.

[11] Kegel, H., and Steimann, F. 2008. Systematically refactoring inheritance to delegation in Java. In Proceedings of the 30th ICSE (May 2008), 431–440.

[12] Ma, Y., Kwon, Y., and Offutt, J. 2002. Inter-class mutation operators for Java. 13th International Symposium on Software Reliability Engineering. IEEE Computer Society Press (November 2002), 352–363.

[13] Offutt J., and Craft, W. 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants. In The Journal of Software Testing, Verification, and Reliability 4, 3 (1994), 131–154.

[14] Offutt, J., Ma, Y., and Kwon, Y. 2006. The class-level mutants of MuJava. In Proceedings of the international workshop on Automation of software test (2006), 78–84.

[15] Offutt, J., and Pan, J. 1997. Automatically detecting equivalent mutants and infeasible paths. In The Journal of Software Testing, Verification, and Reliability 7, 3 (1997), 165–192.

[16] Offutt, J., and Pan, J. 1996. Detecting equivalent mutants and the feasible path problem. In Proceedings of the 11th Conference on Computer Assurance (1996), 224–236.

[17] Schäfer, M., Ekman T., and de Moor, O. 2008. Sound and extensible renaming for Java. In Proceedings of the OOPSLA (2008), 277–294.

[18] Schuler, D., Dallmeier, V., and Zeller, A. 2009. Efficient mutation testing by checking invariant violations. In Proceedings of the 18th international Symposium on Software Testing and Analysis (2009), 69–80.

[19] Steimann, F., and Thies, A. 2009. From public to private to absent: Refactoring Java programs under constrained accessibility. In Proceedings of the 23rd European conference on Object-Oriented Programming (2009), 419–443.

[20] Tip, F., Kiezun, A., and Bäumer, D. 2003. Refactoring for generalization using type constraints. In Proceedings of the OOPSLA (2003), 13–26.

[21] Tip, F. 2007. Refactoring using type constraints. In Proceedings of the 14th International Static Analysis Symposium (2007), 1–17.