Bytecode Testability Transformation

Yanchuan Li and Gordon Fraser

Saarland University, Saarbruecken-66123, Germany
yanchuan@st.cs.uni-saarland.de, fraser@cs.uni-saarland.de

Abstract. Bytecode as produced by modern programming languages is well suited for search-based testing: Different languages compile to the same bytecode, bytecode is available also for third party libraries, all predicates are atomic and side-effect free, and instrumentation can be performed without recompilation. However, bytecode is also susceptible to the flag problem; in fact, regular source code statements such as floating point operations might create unexpected flag problems on the bytecode level. We present an implementation of state-of-the-art testability transformation for Java bytecode, such that all Boolean values are replaced by integers that preserve information about branch distances, even across method boundaries. The transformation preserves both the original semantics and structure, allowing it to be transparently plugged into any bytecode-based testing tool. Experiments on flag problem benchmarks show the effectiveness of the transformation, while experiments on open source libraries show that although this type of problem can be handled efficiently it is less frequent than expected.

1 Introduction

Search-based testing can efficiently generate test inputs that trigger almost any desired path through a program. At the core of these techniques is the fitness function, which estimates how close a candidate solution comes to satisfying its objective. Traditionally, this fitness is based on distances in the control flow and distance estimates for predicate evaluation. The latter are sensitive to Boolean flags, in which the distance information is lost on the way to the target predicate, thus giving no guidance during the search.

Traditionally, search-based testing requires that the source code of the program under test (PUT) is instrumented to collect information required for the distance estimation during execution. The instrumented program is compiled and repeatedly executed as part of fitness evaluations. The fitness evaluation is hindered by problems such as Boolean flags, in which information that could be used for fitness guidance is lost. Testability transformation [8] has been introduced as a solution to overcome this problem, by changing the source code such that information lost at flag creation is propagated to the predicates where flags are used.

If Boolean flags are created outside the scope of the PUT, the source code for these might not be available (e.g., third party libraries), traditional testability transformation is not possible. In contrast, languages based on bytecode interpretation such as Java or C# have the advantage that the bytecode is mostly available even for third party libraries (except for some cases of calls to native code). Bytecode is well suited for search based testing: Complex predicates in the source code are compiled to atomic predicates based

M.B. Cohen and M. Ó Cinnéide (Eds.): SSBSE 2011, LNCS 6956, pp. 237–251, 2011.

on integers in the bytecode. These atomic predicates are always side-effect free, and instrumenting the bytecode to measure branch distances at these predicates is straight forward. In addition, bytecode instrumentation can be done during class loading or even in memory, thus removing the need to recompile instrumented code.

In this paper, we present a bytecode testability transformation which allows us to retain the information traditionally lost when Booleans are defined, thus improving the guidance during search-based testing. In detail, the contributions of this paper are:

Bytecode Instrumentation: Based on previous work in testability transformation [15], we present a semantics preserving transformation of bytecode, which improves the search landscape with respect to traditional Boolean flags as well as those introduced during the compilation to bytecode.

Testability Transformation for Object Oriented Code: The transformation is interprocedural, preserving the information across method calls and interfaces. In addition, the transformation applies to object-oriented constructs, transforming all class members, while preserving validity with respect to references to and inheritance from non-transformable classes (e.g., `java.lang.Object`).

Evaluation: We apply the transformation to a set of open source libraries, thus allowing us to measure the effects of the flag problem in real world software.

This paper is organized as follows: First, we give all the necessary details of search-based testing based on bytecode (Section 2). Then, we describe the details of our transformation in Section 3. Finally, we present the results of evaluating the transformation on a set of case study examples and open source libraries in Section 4.

2 Background

2.1 Search-Based Testing

Search-based testing applies efficient meta-heuristic search techniques to the task of test data generation [10]. For example, in a genetic algorithm a population of candidate solutions (i.e., potential test cases) is evolved towards satisfying a chosen coverage criterion. The search is guided by a fitness function that estimates how close a candidate solution is to satisfying a coverage goal.

The initial population is usually generated randomly, i.e., a fixed number of random numbers for the input values is generated. The operators used in the evolution of this initial population depend on the chosen representation. For example, in a bitvector representation, crossover between two individuals would split the parent bitvectors at random positions and merge them together, and mutation would flip bits.

A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found. For example, to generate tests for branch coverage a common fitness function [10] integrates the *approach-level* (number of unsatisfied control dependencies) and the *branch distance* (estimation of how close the deviating condition is to evaluating as desired).

In this paper we consider object oriented software, for which test cases are essentially small programs exercising the classes under test. Search-based techniques have been applied to test object oriented software using method sequences [3, 7, 13] and strongly typed genetic programming [12, 16].

2.2 Bytecode and Bytecode Instrumentation

Modern programming languages such as Java or those of the .NET framework do not follow the traditional process of compilation to machine code, but are compiled to an intermediate format (bytecode) which is interpreted by virtual machines. The main advantage of such an approach is that the same bytecode can be executed on any platform for which there is a virtual machine available. In addition, it is possible to compile source code of different languages to the same bytecode: For example, all .NET languages (e.g., C# or VB) compile to the same bytecode, and many languages such as Ada, Groovy or Scala can be compiled to Java bytecode. Although machine independent, bytecode is traditionally very close to machine code while retaining some of the information traditionally only available in the source code. As such, it is well suited for different types of analyses even when source code is not available.

An important feature of languages that are based on interpreting bytecode is that they conveniently allow manipulation of the bytecode during class loading, such that instrumentation can be performed without recompilation. In addition, bytecode is at a lower level of abstraction, where the choice of different bytecode instructions is usually smaller than the possible syntactic constructs at source code level, thus making analysis much simpler.

In this paper, we focus on the Java language and bytecode. A detailed description of Java bytecode is out of the scope of this paper; we give the details necessary to understand the transformation, and refer the interested reader to the specification of the Java virtual machine [9]. Java bytecode is based on a stack machine architecture, which retains the information about classes and methods. Each method is represented as a sequence of bytecode instructions, where dedicated registers represent the method parameters and the special value `this`.

The most interesting aspect for search-based testing is that all predicates in source code are translated to simple but potentially nested jump conditions in the bytecode. These conditions operate only integer values, and are free of side effects. Each jump condition consists of an op-code that denotes the type of condition, and a target label. If the condition evaluates to true, then execution jumps to the position in the instruction sequence labelled with the target label, else it proceeds with the next bytecode instruction in sequence. There are different categories of jump conditions; for example, Table 1 lists the conditional jump instructions that compare integer values. Each of the operations in the left half of the table pops a value from the stack and compares it to 0. Similar operations are available to compare identity of object references (IF_ACMPEQ, IF_ACMPNE) and comparison of an object reference to the special value null (IF_NULL, IF_NONNULL). Finally, there is also an unconditional jump operation (GOTO), which always jumps to the target label.

The Java API provides an instrumentation interface, where each class is passed on to different instrumentation classes when loaded. There are several libraries available which allow this instrumentation to be done very conveniently. In our experiments, we used the library ASM[1].

A straight forward approach to search-based testing is to instrument the target program with additional calls that track information about the control flow and branch

[1] `http://asm.ow2.org/`

Table 1. Branch instructions in Java bytecode based on integer operators; top denotes the top value on the stack, top' denotes the value below top

Operator	Description	Operator	Description
IFEQ	$top = 0$	IF_ICMPEQ	$top' = top$
IFNE	$top \neq 0$	IF_ICMPNE	$top' \neq top$
IFLT	$top < 0$	IF_ICMPLT	$top' < top$
IFLE	$top \leq 0$	IF_ICMPLE	$top' \leq top$
IFGT	$top > 0$	IF_ICMPGT	$top' > top$
IFGE	$top \geq 0$	IF_ICMPGE	$top' \geq top$

distances — such instrumentation can easily be done at the bytecode level. For example, our recent EvoSuite [6] prototype adds a method call before each conditional branch in the bytecode, which keeps track of the top elements on the stack and the op-code of the branch instruction, thus allowing the calculation of precise fitness values.

2.3 Testability Transformation

The success of search-based testing depends on the availability of appropriate fitness functions that guide towards an optimal solution. In practice, the search landscape described by these fitness functions often contains problematic areas such as local optima, i.e., candidate solutions may have better fitness than their neighbors but are not globally optimal, thus inhibiting exploration. Another problem are plateaux in the search landscape, where individuals have the same fitness as their neighborhood, which lets the search degrade to random search. A typical source of such problems are Boolean flags or nested predicates, and a common solution is testability transformation [8], which tries to avoid the problem by altering the source code in a way that improves the search landscape before applying the search.

Harman et al. [8] categorize different instances of the flag problem and present transformations to lift instances to easier levels, until the flag problem disappears at level 0. For example, a flag problem of level 1 defines a Boolean flag (`boolean flag = x > 0;`) and then uses the flag (`if(flag) ...`) without any computation on the flag in between definition and use. In its original form, this transformation only works in an intraprocedural setting, and the structure of the program may be changed.

Recently, Wappler et al. [15, 17] presented a solution for function assigned flags. This technique consists of three different tactics: branch completion, data type substitution, and local instrumentation. Data type substitution replaces Boolean values with floating point variables, where positive values represent true and negative values represent false, and these values are calculated by the local instrumentation. Our approach applies these tactics, and extends the approach to apply to bytecode instrumentation.

In this paper, we are mainly focusing on the problem of Boolean flags. Testability transformation has been successfully applied to solve other related problems. For example, a special case of the flag problem is when Boolean flags are assigned within loops [4, 5] and nested predicates [11] can cause local optima even when there are no Boolean flags.

3 Bytecode Testability Transformation

The idea of bytecode testability transformation is to transform bytecode during load-time, such that the information loss due to Booleans is reduced. Ideally, we want this transformation to be transparent to the user, such that the transformation can be plugged into any search-based testing tool without requiring any modifications. In particular, this means that the transformation should not introduce new branches in the source code. While testability transformation as it was defined originally [8] explicitly allows that the semantics of the program are changed by the transformation, as long as the resulting test cases apply to the original program, we want our transformation to preserve the original semantics.

3.1 Boolean Flags In Bytecode

In general, a flag variable is a Boolean variable that results from some computation such that information is necessarily lost. In Java bytecode, there is no dedicated Boolean datatype, but Booleans are compiled to integers that are only assigned the values 0 (ICONST_0 for false) and 1 (ICONST_1 for true). The typical pattern producing such a flag looks as follows:

```
boolean flag = x <= 0;
```

```
L0:
 IFLE L1
 ICONST_0
 GOTO L2
L1:
 ICONST_1
L2:
 // ...
```

```
boolean flag = x > 0;
```

```
L0:
 IFLE L1
 ICONST_1
 GOTO L2
L1:
 ICONST_0
L2:
 // ...
```

It is interesting to note that even though there is no branch here in the source code, at bytecode level we do have a branching instruction when defining a Boolean flag. When such a flag is used in a predicate, this predicate checks whether the flag equals to 0 (IFEQ) or does not equal to 0 (IFNE):

```
if(flag)
   // some code
```

```
L0:
 IFEQ L1
 // some code
 GOTO L2
L1:
 // flag is false
L2:
 // ...
```

```
if(!flag)
   // some code
```

```
L0:
 IFNE L1
 // some code
 GOTO L2
L1:
 // flag is true
L2:
 // ...
```

These examples show how flags are defined and used, but much of the difficulty of Boolean flags at bytecode level arises from how the Booleans are propagated from their definition to their usage. In the simplest case, the Boolean flag would be stored in a register for a local variable (ISTORE), and then loaded immediately before usage (ILOAD). However, Boolean values may also be passed via method calls (e.g., INVOKEVIRTUAL, INVOKESTATIC) or via fields (e.g., SETSTATIC, SETFIELD), or they may not be stored explicitly at all but simply exist on the operand stack.

3.2 Testability Transformation

The general principle of our transformation is similar to that presented by Wappler et al. [17]: We replace Boolean variables with values that represent "how" true or false a particular value is. In Java bytecode, all (interesting) branching operations act on integers, and we therefore replace all Boolean values with integers. Positive values denote true, and the larger the value is, the "truer" it is. Negative values, on the other hand, denote different grades of false. We further define a maximum value K, such that a transformed Boolean is always in the range $[-K,K]$.

When a flag is defined, we need to keep track of the distance value that the condition creating the flag represents, such that this value is used instead of the Boolean value for assignment to a local variable, class variable, as a parameter, or anonymously (e.g., if the flag usage immediately follows the definition). To achieve this, the transformation consists of two parts: First, we have to keep track of distance values at the predicates where Boolean flags are created, and second we need to replace Boolean assignments with integer values based on these distance values.

To keep track of distance values, we insert method calls before predicate evaluation as follows:

```
L0:
 IFLE L1
 // false branch
 GOTO L2
L1:
 // true branch
L2:
 // ...
```

```
L0:
 DUP
 INVOKESTATIC push
 IFLE L1
 // false branch
 GOTO L2
L1:
 // true branch
L2:
// ...
```

The special method `push` keeps a stack of the absolute values of the distance values observed, as predicates can be nested in the bytecode. This way, the top of the stack will always contain the most recently evaluated predicate, and will also tell us how many predicates were evaluated on the way to this predicate. The distance of a predicate essentially equals the distance between the two elements of the comparison. In the case of comparisons to 0, we therefore have to duplicate the top element on the stack (`DUP`), and this value already represents $top - 0$. For comparisons of two integer values, we have to duplicate the top two elements (`DUP2`) and then calculate their difference

(ISUB), which is then passed to the method (push). Another reason why we need the value stack is that all conditions in bytecode are atomic – even simple conjunctions or disjunctions in bytecode are compiled to nested predicates.

As a method may call other methods after which execution returns to the first method, each method has its own such stack. This essentially means there is a stack of stacks: Each time a method is called, a new value stack is put on this stack, and when the method is left via a return or throw statement, the value stack is removed again.

To complete the transformation, the distance values need to be checked when a Boolean value is assigned. To achieve this, we insert a call to the GETDISTANCE function (see Algorithm 1), which is a variant of the method used by Wappler et al. [15], before an assignment to a Boolean variable, i.e., whenever a Boolean value is assigned to a local variable (ISTORE), a Boolean field value (PUTSTATIC, PUTFIELD), used as a Boolean parameter of a method call (INVOKEVIRTUAL, INVOKESTATIC), or used as return value of a method. This function takes the Boolean value resulting from the flag definition, and replaces it with an integer value based on the distance of the last predicate evaluation. GETDISTANCE creates a normalized value in the range $[0,1]$, and scales it across the range $[0,K]$. If the original value was *false*, then the value is multiplied with -1. The call is inserted at the end of a nested predicate evaluation, and the stack depth represents how far evaluation in the predicate has evaluated.

```
L0:
 // Flag definition
 IFNE  L1
 ICONST_1
 GOTO  L2
L1:
 ICONST_0
L2:
 // Store flag
 ISTORE 1
```

```
L0:
 // Flag definition
 IFNE  L1
 ICONST_1
 GOTO  L2
L1:
 ICONST_0
L2:
 INVOKESTATIC getDistance
 ISTORE 1
```

Whenever a Boolean flag is used in a branch condition (IFNE or IFEQ), we have to replace the comparison operators acting on transformed values to check whether the value is greater than 0 or not (IFGT/IFLE).

```
// load flag
 ILOAD 0
 IFEQ L1
 // flag is true
 // ...
 GOTO  L2
L1:
 // flag is false
 // ...
L2:
 // ...
```

```
// load transformed flag
 ILOAD 0
 IFLE L1
 // flag is true
 // ...
 GOTO  L2
L1:
 // flag is false
 // ...
L2:
 // ...
```

Algorithm 1. Get distance value

Require: Boolean value $orig$
Require: Predicate distance stack $stack$
Ensure: Transformed Boolean value d

```
1: procedure GETDISTANCE(orig)
2:     if stack is empty then
3:         distance ← K
4:     else
5:         distance ← stack.pop()
6:     end if
7:     d ← K × (1.0 + normalize(distance))/2^(size of stack)
8:     if orig ≤ 0 then
9:         d ← −d
10:    end if
11:    stack.clear()
12:    return d;
13: end procedure
```

When a Boolean value is negated, in bytecode this amounts to a branching structure assigning true or false depending on the value of the original Boolean. This case is automatically handled by the already described transformations.

There are some branch conditions that do not operate on integers (see Table 2). While these operators themselves do not need to be transformed, they are part of the branching structure and we therefore add their representative truth values on to the value stack. In principle, this amounts to adding either $+K$ or $-K$, depending on the outcome of the comparison.

Table 2. Non-integer comparisons

Operator	Description
IF_ACMPEQ	Top two references on the stack are identical
IF_ACMPNE	Top two references on the stack are not identical
IF_NULL	Top value on stack equals null reference
IF_NONNULL	Top value on stack does not equal null

Furthermore, there is the `instanceof` operation that checks whether an object is an instance of a given class, and returns the result of this comparison as a Boolean. We simply replace any instanceof operations with calls to a custom made call that returns $+K$ or $-K$ depending on the truth value of `instanceof`.

Another type of operator that needs special treatment as an effect of the transformation are bitwise operators (arithmetic operations on Booleans are not allowed by the compiler): For example, a bitwise and of Boolean true (1) and false (0) is false (0), whereas a bitwise and of a negative and a positive integer might very well return a number that is not equal to 0. We therefore have to replace bitwise operations performed on transformed Booleans using replacement functions as follows: A binary AND (IAND) of two transformed Booleans returns the minimum of the two values; A binary XOR

(IXOR) of two transformed values a, b returns $-|a-b|$ if both a and b are greater than 0 or both are smaller than 0, else it returns maximum of a and b (i.e., the positive number). Finally, a binary OR (IOR) returns the largest positive number if there is at least one, or the smallest negative number in case both values are negative.

3.3 Instrumenting Non-integer Comparisons

Except for those listed in Table 2, branch instructions in Java bytecode are exclusively defined on integers. Non-integer variables (long, float, double) are first compared with each other (using the operators LCMP, DCMPL, DCMPG, FCMPL, FCMPG) and the result is stored as an integer -1, 0, or 1 representing that the first value is smaller, equal, or larger than the second value of the comparison. This integer is then compared with 0 using standard operators such as IFLE. This is also an instance of a flag problem, as the branch distance on the branching predicate gives no guidance at all to the search.

To avoid this kind of flag problem, we replace the non-integer comparison operator with an operator to calculate the difference (DSUB, LSUB, FSUB), and then pass the difference of the operators on to a function (`fromDouble`) that derives an integer representation of the value:

```
DLOAD 1
DLOAD 2
DCMPL
IFLE L1
// ...
```

```
DLOAD 1
DLOAD 2
DSUB
INVOKESTATIC fromDouble
IFLE L1
// ...
```

When calculating the integer representation one has to take care that longs and doubles can be larger than the largest number representable as an integer (usually, an integer is a 32 bit number, while longs and doubles are 64 bit numbers). In addition, for floats and doubles guidance on the decimal places is less important the larger the distance value is, but gets more important the smaller the distance value is. Therefore, we normalize the distance values in the range $[0,1]$ using the normalization function $x = x/(x+1)$, which does precisely this (cf. Arcuri [1]), and then multiply the resulting floating point number x with the possible range of integer values. This means that the `fromDouble` function returns $(int)round(K * signum(d) * abs(d)/(1.0 + abs(d)))$ for the difference d.

3.4 Instrumenting Interfaces

Object oriented programs generally follow a style of many short methods rather than large monolithic code blocks. This means that very often, flags do not only exist within a single method but across method boundaries. As we are replacing Boolean flags with integer values, we also have to adapt method interfaces such that the transformation applies not only in an intra-method scenario, but also in an inter-method scenario.

To adapt the interfaces, we have to change both field declarations and method and constructor signatures. There is a possibility that changing the signature of a method

results in a conflict with another existing method in the case of method overloading. If such a conflict occurs, then in addition to the signature the method name is also changed. Furthermore, in addition to the interface declarations every single call to a transformed method or access to a transformed field in the bytecode has to be updated to reflect the change in the signature or name.

It is important to ensure that the transformation is also consistent across inheritance hierarchies, such that overriding works as expected. However, there are limits to the classes and interfaces that can be transformed: Some base classes are already loaded in order to execute the code that performs the bytecode transformation. In addition, it might not be desirable to instrument the code of the test generator itself. Therefore we only instrument classes that are in the package that contains the unit under test, or any other user specified packages.

This, however, potentially creates two problems: First, some essential interfaces define Boolean return values and are used by all Java classes. For example, the `Object.equals` class cannot be changed, but is a potential source of Boolean flags. Second, a called method might receive a transformed Boolean value as parameter, but expects a real Boolean value. In these cases, a transformed Boolean is transformed back to a normal Boolean value representing whether the transformed value is greater than 0 or not, such that the normal Boolean comparisons to 0 and 1 work as expected. Similarly, we have to transform Boolean values received from non-transformed methods and fields back to the integer values $+K$ or $-K$.

3.5 Instrumenting Implicit Else Branches

Often, a Boolean value is only assigned a new value if a predicate evaluates in one way, but not if it evaluates the other way. In this case, if the value is not assigned, we have no guidance on how to reach the case that the condition evaluates to the other value. To overcome this problem, we add implicit else branches; this technique is referred to as *branch completion* by Wappler et al. [17], who introduced it to ensure that a guiding distance value can always be calculated.

```
  ILOAD 0
  IFLE L1
  ICONST_0
  ISTORE 1
L1:
  // ...
```

```
  ILOAD 0
  IFLE L1
  ICONST_0
  ISTORE 1
  GOTO L2
L1:
  ILOAD 1
  INVOKESTATIC GetDistance
  ISTORE 1
L2:
  // ...
```

We add such an implicit else branch whenever a Boolean value is assigned to a field or a local variable (PUTSTATIC, PUTFIELD, ISTORE), such that we can easily add the else branch. In the example, the value is assigned to local variable 1, therefore in the

implicit else branch we first load this variable (`ILOAD 1`) and then store a transformed Boolean value based on the current predicate (`GetDistance`) again (`ISTORE 1`).

4 Evaluation

We have implemented the described bytecode transformation as part of our evolutionary test generation tool EvoSuite [6]. To measure the effects of the transformation with as little as possible side-effects, we ignored collateral coverage in our experiments, i.e., we only count a branch as covered if EvoSuite was able to create a test case with this branch as optimization target. We deactivated optimizations such as reusing constants in the source code, and limited the range of numbers to $\pm 2{,}048{,}000$. EvoSuite was configured to use a (1+1)EA search algorithm, for details of the mutation probabilities and operators please refer to [6]. EvoSuite was further configured to derive test cases for individual branches, such that individuals of the search equal to sequences of method calls. The length of these sequences is dynamic, but was limited to 40 statements. Each experiment was repeated 30 times with different random seeds; to allow a fair comparison despite the variable length of individuals we restricted the search budget in terms of the number of executed statements.

4.1 Flag Problem Examples

To study the effects of the transformation, we first use a set of handwoven examples that illustrate the effectiveness of the transformation, and run test generation with a search limit of $300{,}000$ statements[2]:

```
// Intra-method flags
class FlagTest1 {

 boolean flag1 = false;

 boolean flagMe(int x) {
  return x == 762;
 }

 void coverMeFirst(int x) {
  if(flagMe(x))
    flag1 = true;
 }

 void coverMe() {
  if(flag1)
   // target branch
 }
}
```

```
// Nested predicates
class FlagTest2 {
 void coverMe(int x,
              int y) {
  boolean flag1 =
        x == 2904;
  boolean flag2 = false;
  if(flag1) {
   if(y == 23598)
    flag2 = true;
  }
  else {
   if(y == 223558)
    flag2 = true;
  }
  if(flag2)
   // target branch
 }
}
```

```
// Example for doubles
// and conjunction
class FlagTest3 {

 void coverMe(double x) {
  if(x > 251.63 &&
     x < 251.69)
   // target branch
 }
}
```

In addition to these three examples, we also use the `Stack` example previously used by Wappler et al. [15] to evaluate their testability transformation approach. The target branch in this example is in the method `add`, which throws an exception if flag method `isFull` returns true. The other examples used by Wappler et al. [15] are in principle also covered by our other examples.

[2] As individuals in EvoSuite are method sequences and can have variable length we count the number of executed statements rather than fitness evaluations.

Table 3. Results of the transformation on case study examples

Example	Without transformation			With transformation		
	Success Rate	Statements	Time/Test	Success Rate	Statements	Time/Test
FlagTest1	0/30	300,001.07	0.06ms	30/30	154,562.40	0.09ms
FlagTest2	0/30	300,001.53	0.06ms	30/30	154,142.37	0.08ms
FlagTest3	0/30	300,000.93	0.07ms	30/30	99,789.53	0.13ms
Stack	0/30	300,001.00	0.07ms	30/30	4,960.40	1.40ms

The first question we want to analyze is whether the transformation has the potential to increase coverage. The four examples clearly show that this is the case (see Table 3): Out of 30 runs, the target branches could not be covered a single time for any of the examples without transformation. With the transformation applied, on the other hand, every single run succeeded, with convergence between 100,000-200,000 executed statements, except for the Stack example which converges already around 5,000 executed statements. This improvement comes at a cost, as can be seen in the average test execution time: The average execution time increases by 34%, 25%, 44%, and 95% for each of the examples, respectively. An average increase of 50% in the execution time can be significant, as every single test case has to be executed as part of the search. Note that our implementation is not optimized in any way, so the 50% increase could likely be reduced by optimizations. However, the question is whether flag problems occur frequently in real software, such that the overhead of the transformation is justified.

4.2 Open Source Libraries

To study whether the potential improvement as observed on the case study subjects also holds on "real" software, we applied EvoSuite and the transformation to a set of open source libraries. We chose four different libraries with the intent to select a wide range of different applications: First, we selected the non-abstract container classes of the java.util library. Furthermore, we selected the non-abstract top level classes of the JDom XML library, and all classes of the Apache Commons Codec and Command Line Interface libraries. We used a search limit of $100{,}000$ executed statements for each branch. Statistical difference has been measured with the Mann-Whitney U test, following the guidelines on statistical analysis described by Arcuri and Briand [2]. To quantify the improvement in a standardized way, we used the Vargha-Delaney $\hat{A}_{12}$ effect size [14]. In our context, the $\hat{A}_{12}$ is an estimation of the probability that EvoSuite with testability transformation can cover more branches than without. When the two types of tests are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ would mean that the testability transformation satisfied more coverage goals in *all* cases.

The results of the analysis are summarized in Table 4. In total, we obtained p-values lower than 0.05 in 36 out of 43 comparisons in which $\hat{A}_{12} \neq 0.5$. In all four libraries, we observed classes where sometimes the transformation seems to decrease the coverage slightly. In particular, this happens when a method takes a Boolean parameter and is therefore transformed to take an integer as input. When mutating a Boolean value, EvoSuite replaces the value with a new random Boolean value. For integers, however,

Table 4. Results of the $\hat{A}_{12}$ effect size on open source libraries. $\hat{A}_{12} > 0.5$ denotes the probability that the transformation leads to higher coverage.

Case Study	Classes	$\#\hat{A}_{12} < 0.5$	$\#\hat{A}_{12} = 0.5$	$\#\hat{A}_{12} > 0.5$	$\varnothing\hat{A}_{12}$
Commons Codec	21	4	11	6	0.53
Commons CLI	14	2	7	5	0.51
Java Collections	16	4	1	11	0.59
JDom	18	5	7	6	0.52
Σ	69	15	26	28	0.53

EvoSuite only replaces the value with a low probability (0.2 in our experiments), but else adds a small (random) delta in the range [-20,20]. It can therefore happen that such a transformed Boolean parameter only sees positive values during the evaluation, while a negative value (i.e., false) would be needed to take a certain branch. We expect that this behavior would disappear for example if the number of generations were increased, or if an algorithm with larger population sizes than the single individual of the (1+1)EA would be used.

In 28 out of 69 cases, the transformation resulted in higher coverage, which is a good result. However, on average over all classes and case study subjects, the $\hat{A}_{12}$ value is 0.53, which looks like only a small improvement. To understand this effect better, we take a closer look at the details of the results. In the Commons Codec library, the coverage with and without the transformation has identical results on 11/21 classes, and only very small variation on the remaining classes except one particular class: `language.DoubleMetaphone` has 502 branches on bytecode, and is the most complex class of the library. With testability transformation, on average 402,5 of these branches are covered; without transformation, the average is 386,8. Testability transformation clearly has an important effect on this class. In the Commons CLI library the picture is similar: On most classes, the coverage is identical or comparable. However, in the `CommandLine` (39.0 out of 45 branches with transformation, 37.6 without) and `Option` (86.3 out of 94 branches with transformation, 85.3 without) classes there seems to be an instance of the flag problem. The Java container classes have several classes where the transformation increases coverage slightly by 1–2 branches each (several `HashMap`, `Hashtable`, and `HashSet` variants. Interestingly, the `Stack` class in the `java.util` library has no flag problem). Finally, JDom also has mainly comparable coverage, with the main exception being the `Attribute` class, which has a clear coverage improvement (50.9 out of 65 branches with transformation, 49.8 without).

In summary, this evaluation shows that the transformation can effectively overcome the flag problem in real-world software — however, the flag problem seems to be less frequent than expected, so when performance is critical, testability transformation might only be activated on-demand when analysis or problems in the search show that there is a flag problem in a class. Potentially, static analysis could be used to identify sections in the bytecode where transformation is necessary, such that the transformation would only need to be applied selectively, thus reducing the overhead of the instrumentation. In general, the increase in the effort may be acceptable as it leads to higher coverage. Furthermore, our evaluation on the open source subjects only considers whether the

coverage has increased or not; the testability transformation might also achieve that branches are covered *faster* than without the transformation, i.e., with fewer iterations of the search algorithm. We plan to investigate this as future work.

5 Threats to Validity

Threats to *construct validity* are on how the performance of a testing technique is defined. Our focus is on the achieved coverage of the test generation. However, our experiments show a clear disadvantage with respect to performance, and we did not evaluate any effects on secondary objectives such as the size of results.

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of defects in our testing framework, it has been carefully tested. In addition, to validate the correctness of our transformation, we used the test suites provided by the open source projects we tested and checked whether the test results were identical before and after the transformation. As in any empirical analysis, there is the threat to *external validity* regarding the generalization to other types of software. We tried to analyze a diverse range of different types of software, but more experiments are definitely desirable.

6 Conclusions

Bytecode as produced by modern languages is well suited for search-based testing, as bytecode is simple, instrumentation is easy and can be done on-the-fly, and predicates are atomic and use mainly integers. However, bytecode is just as susceptible to the flag problem as source code is. In fact, the compilation to bytecode even adds new sources of flags that need to be countered in a transformation. In this paper, we presented such a transformation, and showed that it can overcome the flag problem.

Experiments showed that the transformation is effective on the types of problems it is conceived for, but it also adds a non-negligible performance overhead. Our experiments on open source software revealed that the flag problem is also less frequent than one would expect, although it can be efficiently handled by the transformation if it occurs. Clearly our experiments in this respect can only be seen as an initial investigation, and further and larger experiments on real software will be necessary to allow any definite conclusions about the frequency of the flag problem. Furthermore, we only analyzed the basic case where the search tries to cover a single target; it is likely that new techniques such as optimization with respect to all coverage goals at the same time, as is also supported by EvoSuite , can affect the flag problem. However, the conclusion we can draw from our experiments is that the presented transformation does overcome the flag problem as expected, but it probably makes most sense to use it on-demand rather than by default, or to identify and focus the transformation only on problematic parts or the program under test.

Acknowledgments. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany.

References

1. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability (2011)
2. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: IEEE International Conference on Software Engineering, ICSE (2011)
3. Arcuri, A., Yao, X.: Search based software testing of object-oriented containers. Information Sciences 178(15), 3075–3095 (2008)
4. Baresel, A., Binkley, D., Harman, M., Korel, B.: Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, pp. 108–118. ACM Press, New York (2004)
5. Binkley, D.W., Harman, M., Lakhotia, K.: Flagremover: A testability transformation for transforming loop assigned flags. ACM Transactions on Software Engineering and Methodology 2(3), 110–146 (2009)
6. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: International Conference On Quality Software, QSIC (2011)
7. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. In: ISSTA 2010: Proceedings of the ACM International Symposium on Software Testing and Analysis, pp. 147–158. ACM, New York (2010)
8. Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. 30, 3–16 (2004)
9. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1999)
10. McMinn, P.: Search-based software test data generation: a survey: Research articles. Software Testing Verification Reliability 14(2), 105–156 (2004)
11. McMinn, P., Binkley, D., Harman, M.: Testability transformation for efficient automated test data search in the presence of nesting. In: Proceedings of the 3rd UK Software Testing Research Workshop (UKTest 2005), Sheffield, UK, September 5-6, pp. 165–182 (2005)
12. Ribeiro, J.C.B.: Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In: GECCO 2008: Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation, pp. 1819–1822. ACM, New York (2008)
13. Tonella, P.: Evolutionary testing of classes. In: ISSTA 2004: Proceedings of the ACM International Symposium on Software Testing and Analysis, pp. 119–128. ACM, New York (2004)
14. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
15. Wappler, S., Baresel, A., Wegener, J.: Improving evolutionary testing in the presence of function-assigned flags. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp. 23–34. IEEE Computer Society Press, Washington, DC, USA (2007)
16. Wappler, S., Lammermann, F.: Using evolutionary algorithms for the unit testing of object-oriented software. In: GECCO 2005: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, pp. 1053–1060. ACM, New York (2005)
17. Wappler, S., Wegener, J., Baresel, A.: Evolutionary testing of software with function-assigned flags. J. Syst. Softw. 82, 1767–1779 (2009)