Guiding Random Test Generation for Intra-Class Dataflow Coverage

Petru Florin Mihancea, Edit Mercedes Mera-Batiz, Marius Minea
LOOSE Research Group
Politehnica University of Timişoara, Romania
Email: petrum, marius@cs.upt.ro

***Abstract*—Automatic generation of a good test suite is difficult, especially for object-oriented software. Feedback-directed random test generation is an approach that can achieve good branch coverage and has been used as a basis to systematically construct suites for testing realistic Java programs. We augment this random test generation method to create tests suites that satisfy an intra-class data-flow coverage criterion which is highly relevant for object orientation, although little addressed or achieved by tools in practice. We show that our approach can be used on real object-oriented software and that the technique for guiding test generation produces an increase in coverage.**

***Keywords*-testing object-oriented programs; dataflow coverage;**

I. Introduction

Generating tests (manually or automatically) is usually subject to satisfying certain coverage criteria. While control-flow criteria are employed more frequently and supported by more coverage measurement tools, dataflow criteria capture a different set of potential errors. In essence, dataflow coverage (in particular, the *all-uses* criterion) aims to check that each use of a variable has been exercised for every possible statement (i.e., distinct computation) that may produce the used value.

Dataflow testing can be particularly effective for object-oriented software, which is characterized by short methods and many method calls which interact. In particular, it can detect various anomalies due to improper method overriding.

The simplest automated test generation approach is random; however, this is not always effective in achieving high coverage. Moreover, it is often difficult to generate meaningful random tests for object-oriented programs, since objects need to be created with certain constraints, and method calls must observe preconditions on data or relative sequencing.

We build on the RANDOOP tool [1] which implements *feedback-directed random test generation*. This means stepwise extension of tests (i.e., sequences of method calls) that run without errors, using objects and arguments constructed by previous method calls, but selecting methods and data at random. Our insight is that guiding the creation of new tests to use objects and methods in view of covering *def-use* pairs has the potential to achieve higher coverage of the desired dataflow criterion. Our contribution is to propose an approach that systematically guides random test generation towards satisfying intra-class dataflow coverage, and to show that it is effective, achieving higher coverage in shorter time.

II. Background

We start by presenting in brief the idea of intra-class dataflow testing [2] followed be the random test generation technique implemented in the RANDOOP tool [1]. These notions will be exemplified based on the code in Listing 1.

Listing 1: Sample Illustrative Java Source Code

```
1  public class SomeClass {
2   private int x;
3   public SomeClass() {
4    x = 0;
5   }
6   public int a() {
7    return x;
8   }
9   public void b() {
10   x = x + 1;
11  }
12  public boolean c(SomeClass y) {
13   return y != null;
14  }
15  public int d() {
16   return 1;
17  }
18 }
```

A. Intra-Class Dataflow Testing

Intra-class dataflow testing [2] is a white-box unit testing method for classes using the notion of *intra-class def-use* pair.

In an intra-class context, a *def* is an assignment of a value to an instance variable of the class. Similarly, a *use* is a program point where the value of an instance variable is accessed. Finally, an *intra-class def-use* pair is defined as a *def* of some value and a successive *use* of that value with respect to the same instance variable, where the *def* and the *use* must be performed in different activations of the involved object. In other words, the *def* and the *use* should be executed as a result of different invocations from a sequence of calls to interface methods of the object/class. For example, in Listing 1, all the *intra-class def-use* pairs are $(def_{\text{line4}} - use_{\text{line7}})$, $(def_{\text{line4}} - use_{\text{line10}})$, $(def_{\text{line10}} - use_{\text{line7}})$ and $(def_{\text{line10}} - use_{\text{line10}})$.

Many different dataflow testing criteria could be used to cover these *def-use* pairs with tests. One of the most practical approaches, as stated in [2], is the *all-uses* criterion, requiring each feasible *def-use* pair to be executed by a test. For the class *SomeClass*, this means the four pairs listed above. Because of its practical usage and simplicity, we also adopt this criterion for our automatic test generation for intra-class testing.

B. Random Test Generation with RANDOOP

In RANDOOP [1] a test s_i is represented as a sequence of method invocations of the form $v_{i,1} = m_{i,1}, v_{i,2} = m_{i,2}, \ldots, v_{i,n} = m_{i,n}$ [1]. The arguments of an invocation $m_{i,k}$ are predefined primitive values, null or references produced by previous invocations in the sequence.

RANDOOP records all sequences produced during the random test generation procedure. More importantly, all tests executed without errors (i.e., observing a set of predefined runtime constraints) are included in a special set *seqs* from where they can be selected for generating other new sequences.

In essence, until a specified time limit is reached, RANDOOP iteratively applies the following procedure:

- a public method from the system is selected randomly
- for each method argument having a primitive type, a value is selected from a predefined set of values
- for each reference argument of the method (including the target object if needed) the tool chooses at random i) to use null ii) to select from the set *seqs* a sequence s_j producing a value $v_{j,k}$ of appropriate argument type or iii) to use a value $v_{j,l}$ from a sequence already selected in step 2 for a previous argument
- finally, all sequences selected to provide values for method arguments are concatenated and the new method invocation is appended to this new sequence. The test is run and, in the conditions mentioned at the beginning of this section, it may be added to the *seqs* set to contribute in turn to generating new longer tests.

As an example, suppose we use RANDOOP to generate tests for the class in Listing 1. In a first iteration of the previous procedure, assume that the method *a* is randomly chosen. Since *seqs* is empty, we cannot find a target object to invoke the method. Consequently, in this iteration, we cannot create any test. In a second iteration, suppose the constructor of the class is selected. Since its invocation does not require any argument, the first sequence is created containing only the constructor invocation (see the first test in Listing 2). Next, the sequence is executed and added to the *seqs* set.

Listing 2: Test Generated with RANDOOP

```
//Test 1
SomeClass var0 = new SomeClass();

//Test 2
SomeClass var0 = new SomeClass();
SomeClass var1 = new SomeClass();
var0.c(var1);
```

Suppose that in the third iteration, method *c* is randomly chosen. RANDOOP searches in the sequences from *seqs* a reference on which to invoke the method and another one to be used as the argument value. The only possible reference is *var0* generated in the previous iteration. Thus, a new test is created by duplicating (once for the target object and once for the argument) the sequence of the first test. Next, the duplicated sequences are concatenated and, at the end, an invocation to method *c* is added (see the second test in Listing 2). Finally, the sequence is executed and added to the *seqs* set. This iterative process continues until an upper time limit is reached.

[1]If a method returns void the assignment is not included

C. The Problem

Although RANDOOP has proved to be efficient in generating tests for object-oriented programs, it does not keep track of any coverage criterion for the resulting tests, nor does it attempt to satisfy any such criterion. The choice of a method to extend a test is purely random. It does not take into consideration what *defs* have been executed by a particular invocation sequence, what methods containing some *uses* should be selected to extend a sequence in order to access the values assigned by the executed *defs*, etc.

Consequently, our idea described subsequently is to guide RANDOOP's method selection, extending the already built tests in a way that increases the likelihood of covering more *intra-class def-use* pairs and achieving better dataflow coverage.

III. GUIDING RANDOOP FOR INTRA-CLASS ALL-USES CRITERION

To guide RANDOOP aiming to produce tests that comply faster to the *all-uses* criterion for *intra-class def-use* pairs, we had to change the normal test generation procedure of the tool, described in II-B and on the left-hand side of Figure 1. In essence, we try to bypass the pure random selection of the method and target object for the creation of a new test. Instead, we keep track of available objects within a test and of the *defs* executed on them and try to extend that test with invocations to methods containing *defs* and/or *uses* not yet executed.

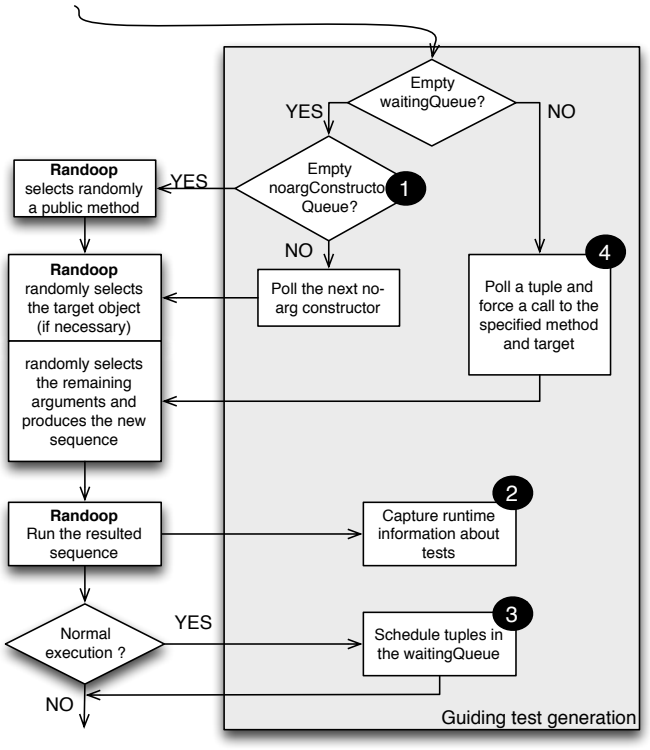


Fig. 1: Integrating the Guiding Procedure in RANDOOP

A. Intra-Class Def-Use Pair Representation

In the guiding procedure that we propose in this paper we represent an *intra-class def-use pair* by a tuple of the form (m_{def}, m_{use}, F, C) in which:

- C is a concrete class from the tested system
- F is a field from class C
- m_{def} is a method of C containing a definition of field F
- m_{use} is a method of C containing a use of field F

Although this is an approximation (i.e., a method may contain several distinct *defs / uses* of the same field) we adopt it because it simplifies both the description and the implementation of the guiding procedure and it is sufficient to evaluate the advantage of our approach over unguided random test generation with respect to *intra-class all-uses* coverage.

B. The Guiding Procedure

The approach maintains a *waitingQueue* of tuples of the form $(s_i, v_{i,j}, (m_{def}, m_{use}, F, C), selector)$ in which:

- s_i is a sequence of method invocations from the tested system (like the sequences produced by RANDOOP)
- $v_{i,j}$ represents a reference value produced by the j^{th} invocation from s_i
- (m_{def}, m_{use}, F, C) represents an *intra-class def-use* pair as previously explained
- *selector* is the symbolic value *forceDef* or *forceUse* meaning that the method m_{def} (respectively, m_{use}) should be invoked on reference $v_{i,j}$

When the *waitingQueue* is not empty, the first tuple is extracted and our guiding procedure tries to produce a new test by forcing the invocation of the corresponding method on the corresponding target according to the tuple description. More details regarding this task are presented in Section III-B4. The reason and the manner in which a new tuple is added to the *waitingQueue* are described in Section III-B3.

On the other hand, when the *waitingQueue* is empty, it means that no tuple awaits processing and control should be returned to the original RANDOOP procedure (i.e., randomly select a method to be invoked). However, we have also added another heuristic that is described in the following.

1) Invoke No-Arg Constructors: Instance variables stay at the heart of an *intra-class def-use* pair. Consequently, we should try to create objects early in the random test generation process. Unfortunately, this is not an easy task: creating an object might require the existence of some other instances to be passed as arguments to the object constructor. To balance these requirements we decided to only enforce the instantiation of the classes having a no-arg constructor.

Consequently, we search for all no-arg constructors in the analyzed program and place them in a *noargConstructorQueue*. When this queue is empty we return control to the original RANDOOP procedure (i.e., randomly select a method to be invoked). However, if the *noargConstructorQueue* is not empty, we extract the first constructor and we pass it to RANDOOP as if it had been randomly chosen.

As an example, consider the code from Listing 1. When the guiding procedure is initialized, the content of the *noargConstructorQueue* is determined. In this case, the single element of the queue is the *SomeClass* constructor. When RANDOOP invokes the test generation procedure, the *waitingQueue* will be empty (because no tuple has been identified yet). However, the *noargConstructorQueue* is not empty and thus, the only constructor is extracted from the queue and returned to RANDOOP as if it had been selected randomly. As a consequence, a first test is immediately created (see Listing 3).

Listing 3: Creating Sequences to Invoke No-Arg Constructors

```
//Test1
SomeClass var0 = new SomeClass();
```

2) Collecting Information About Test Execution: An essential step in the RANDOOP test generation procedure is to run the generated test in order to identify tests that could be extended with invocations to new methods. We take advantage of this execution to identify runtime information such as:

- for each object produced during a test execution, we determine which method m_{def} executed the last definition of a field F of that instance of class C
- when a *use* is detected for a field F in an instance of class C, we determine the m_{use} method performing the access; next, based on the previous information, we identify the method m_{def} performing the last definition of F from class C on the same object; consequently, in a simplified manner, we can determine which (m_{def}, m_{use}, F, C) pair has been covered
- for each object produced during a test execution, we record (in their execution order) a list of methods performing a *def* or a *use* of some field of the object

This information is used further to guide the test generation procedure and is obtained by instrumenting the tested code. In essence, we capture the execution of bytecode instructions related to the definition and usage of a field together with the method performing the access and the target object. Based on this primary information we can derive all the previously mentioned knowledge regarding the execution of the tests.

As an example, after the execution of the single test from Listing 3, our guiding procedure will record that the last definition of the field x from class *SomeClass* has been performed by the m_{def} method (more precisely, constructor) *SomeClass*. Second, it will be also recorded that no *intra-class def-use* pair has been covered by the test. Finally, it will be determined that, on the created instance, only the initialization operation has performed a definition of some field.

3) Scheduling Enforced Method Invocations: As mentioned earlier, our idea is to try to bypass the random method selection performed by RANDOOP to create a new test. For this purpose, we maintain a queue of tuples containing information about interesting target objects and methods to be invoked in order to cover more *intra-class def-use* pairs. In this section we show how we determine these interesting instances and methods.

Basically, after a successful execution of a newly generated test s_k of the form $v_{k,1} = m_{k,1}, v_{k,2} = m_{k,2}, ..., v_{k,n} = m_{k,n}$

we analyze the collected runtime information and add new elements to the *waitingQueue* using the following two steps.

a) Try Forcing a Use: As we have mentioned, each *intra-class def-use pair* is represented as a tuple of the form (m_{def}, m_{use}, F, C). Clearly, this information can be computed statically for all classes of the tested program.

Assume now that the test s_k has executed successfully. As explained in Section III-B2, for each object created during a test execution, we know what method performed the last definition of each field of that object. Consequently, for an object of class C referred by a reference $v_{k,l}$ from s_k we know the method m_{def} that performed the last definitions of the field F. Thus, for the object $v_{k,l}$ of class C we can try to force an invocation to any method m_{use} of C containing a use of the field F. In other words, we try to invoke a method m_{use} that might access the last value set for F and thus cover the *intra-class def-use* pair (m_{def}, m_{use}, F, C).

As a result of this step, several tuples having the form $(s_k, v_{k,l}, (m_{def}, m_{use}, F, C), forceUse)$ will be added to the *waitingQueue*. Continuing with the example based on the class from Listing 1, as explained at the end of the previous section, by executing the single test available until now (see Listing 3), the last definition of field x on the instance $var0$ of *SomeClass* has been performed by the class constructor. It is easy to see that all def-use pairs in which the first method contains a definition of field x and the second method contains a use of that field are: $(SomeClass, a, x, SomeClass)$, $(SomeClass, b, x, SomeClass)$, $(b, a, x, SomeClass)$, and $(b, b, x, SomeClass)$. Thus, in order to cover new *intra-class def-use* pairs, we should try to generate a test in which we invoke method a on $var0$ and another one in which we should try to invoke method b on the same reference. Consequently, two tuples are added to the *waitingQueue*: $(test1, var0, (SomeClass, a, x, SomeClass), forceUse)$ and $(test1, var0, (SomeClass, b, x, SomeClass), forceUse)$.

The execution of this selection step is controlled by several constraints described in the following:

1) a tuple is added to the *waitingQueue* if and only if the corresponding (m_{def}, m_{use}, F, C) is not yet covered; the rationale is that if an *intra-class def-use* pair has been covered we should not try to cover it again;
2) if the executed sequence s_k has been obtained by our guiding procedure, a new tuple is added to the *waitingQueue* only if the sequence was not created by the same *Try Forcing a Use* step. The reason is that part of sequence s_k has been already used by this step, and reapplying the step will likely duplicate tests. However, there is an exception from this constraint: if s_k has covered new *intra-class def-use* pairs, we allow reapplying the same step because some *uses* might become reachable (e.g., an *use* in an object may alter the state of an object component, enabling the execution of another *use* in that object).

b) Try Forcing a Def: While the purpose of the previous step is to try to exercise invocations of methods performing new *uses* of some fields of the target objects, the purpose of the second step is to call methods in the hope of executing new definitions of some fields.

As mentioned, we can statically compute all tuples (m_{def}, m_{use}, F, C) representing the *intra-class def-use* pairs for every class C. Moreover, during the execution of our modified random test generation, we know which tuples (m_{def}, m_{use}, F, C) have already been covered. Consequently, for each reference $v_{k,l}$ from a sequence s_k pointing to an object of type C, we can easily determine which tuples (m_{def}, m_{use}, F, C) have not been covered yet and thus, which methods m_{def} we should try to invoke on the object $v_{k,l}$.

As a result of this step, several tuples of the form $(s_k, v_{k,l}, (m_{def}, m_{use}, F, C), forceDef)$ are added to the *waitingQueue*. Recall that the *intra-class def-use* pairs for Listing 1 are $(SomeClass, a, x, SomeClass)$, $(SomeClass, b, x, SomeClass)$, $(b, a, x, SomeClass)$, and $(b, b, x, SomeClass)$. Moreover, as shown at the end of section III-B2, during the execution of the single test available so far (see Listing 3), we have a single object of class *SomeClass* referred by $var0$ and no *intra-class def-use* pair has been covered yet. To increase the likelihood of covering these pairs we should try to execute the definitions of these pairs. Thus, we should generate tests which invoke method *b* on $var0$[2]. As a result, the following tuples are added to the *waitingQueue*: $(test1, var0, (b, a, x, SomeClass), forceDef)$ and $(test1, var0, (b, b, x, SomeClass), forceDef)$.

Like the previous step, forcing a *def* is also controlled by several constraints described in the following:

1) a tuple is added to the *waitingQueue* if and only if the corresponding (m_{def}, m_{use}, F, C) has not been already covered; the reason is that we should not force another execution of the same *intra-class def-use* pair since it has been already executed;
2) if the executed sequence s_k has been obtained by our guiding procedure we do not permit the execution of the *Try Forcing a Def* step; the reason is that s_k may already contain a sub-sequence that has been already processed by this step and thus, reapplying it would result in highly duplicated tests;
3) a tuple is added to the *waitingQueue* if and only if the corresponding (m_{def}, m_{use}, F, C) is not already present in another tuple from the *waitingQueue*; the primary rationale is that the test might have performed the *def* from m_{def} but not also the *use* from m_{use}. In this case, forcing an invocation to m_{def} is not needed since only a *use* should be forced trying to cover the pair. However, forcing the m_{use} invocation has already been done by the *Try Forcing a Use* step.

4) Produce a Guided Test: As we have mentioned at the beginning of the guiding procedure description, when RANDOOP starts trying to generate a new test, we try to bypass the random selection of a method to be invoked. For that purpose, the *waitingQueue* is fed with information as described in the

[2]Since a constructor can be executed only at the object creation time, the def methods from the first two tuples are ignored in this step

previous paragraphs. If the *waitingQueue* is empty, we usually let RANDOOP produce a test in its normal manner[3]. Otherwise, the first tuple from the queue is processed. We remind that this tuple has the form $(s_i, v_{i,j}, (m_{def}, m_{use}, F, C), selector)$.

The *selector* has the symbolic values *forceDef* or *forceUse* and is used to choose which method to invoke: m_{def} or m_{use}. The target reference of the invocation will be $v_{i,j}$ from the sequence s_i. For the remaining arguments of the invocation we perform exactly the same actions as RANDOOP, e.g., for a reference parameter, a compatible value $v_{x,y}$ from the test s_x is randomly chosen as the actual value of the parameter. Finally, all selected tests/sequences are concatenated (including s_i) and the invocation to m_{def}/m_{use} on $v_{i,j}$ is appended at the end. As can be seen, a test is produced in a very similar way to RANDOOP but we explicitly mention the invoked method and the target reference to increase the likelihood of covering *intra-class def-use* pairs according to the *all-uses* criterion.

To exemplify the generation of a new test, we return to the code in Listing 1. While explaining the guiding procedure, we have seen that after the execution and analysis of the test in Listing 3, the *waitingQueue* contains the following tuples:
$(test1, var0, (SomeClass, a, x, SomeClass), forceUse)$,
$(test1, var0, (SomeClass, b, x, SomeClass), forceUse)$,
$(test1, var0, (b, a, x, SomeClass), forceDef)$ and
$(test1, var0, (b, b, x, SomeClass), forceDef)$.
The first tuple is extracted and the first test (i.e., Test1) is duplicated to provide a reference (e.g., $var0$) on which to invoke method *a* (i.e., the method containing an use that should be executed). Since the method has no additional arguments, the new test (i.e., Test2) can be produced and it is shown in Listing 4[4]. Here, the execution of the second test immediately covers an *intra-class def-use* pair: the definition of the instance variable *x* from the constructor and its use inside method *a*.

Listing 4: Producing a New Test

```
//Test1
SomeClass var0 = new SomeClass();

//Test2
SomeClass var0 = new SomeClass();
int var1 = var0.a();
```

To avoid generating unnecessary tests we add some constraints that guard the test construction:

1) If all *intra-class def-use* pairs of class C corresponding to the current working tuple have been covered, the test construction is useless and no test is produced;
2) As mentioned in Section III-B2, for each instances we record the list of operations performing a *def* or an *use* to some field of the object. Consequently, when we force a call to a method m_{def} / m_{use} on an object, we can check to see if the resulting sequence of invocations have not already been performed on another object of the same class. If such an object exists, then we do not create a new test. The reason is that the two objects might have the same state and thus, applying our guiding procedure on them would likely have the same effect, resulting in duplicated tests. Moreover, if our guiding procedure fails to cover some *def-use pair* using an object, reapplying the procedure on an object with a similar state would probably also fail to cover that *def-use pair*.

[3]The single exception is described in Section III-B1

[4]We mention that RANDOOP eliminates subsumed tests and thus the final jUnit test suite will not contain the first test

IV. EVALUATION

To evaluate the proposed approach we have created a modified version of RANDOOP that includes our guiding technique. For this, we have used the ASM framework [3] and the CODEPRO analysis tool [4]. In this section we describe the conducted case study and we discuss the results.

A. Scenario

For our evaluation we have selected the JUNG network/graph framework[5], implemented in Java. From this system we have eliminated some irrelevant classes that are usually not targeted by testing activities (e.g., the implementations of other tests, etc.). Moreover, since we aim to interact with concrete objects, we have prepared for testing with the original/modified RANDOOP tool only the concrete non-nested/non-inner accessible classes. Consequently, 250 classes from the JUNG system remained to be used for our evaluation.

The main goal of our case study was to establish if the proposed guiding technique can improve the *intra-class def-use pair* coverage compared to the random test generation technique implemented in RANDOOP, and whether the obtained coverage increases faster. Consequently, we have generated tests using i) the original RANDOOP and ii) the modified RANDOOP including the proposed guiding approach. For each test suite obtained, we have estimated the *intra-class def-use* pair coverage and we have compared the results.

This process has been repeated for different time execution limits. In each execution we used the default RANDOOP options including the randomization seed (the single exceptions being the time limit and the list of tested classes as previously described). Moreover, to limit the influence of non-determinism[6], we have repeated each execution 3 times and we have reported and compared the averages.

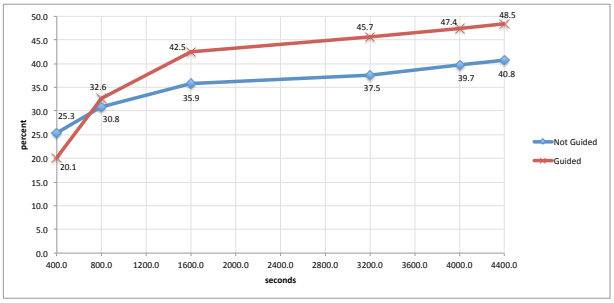
To ensure a proper comparison, we have included in the maximum execution time the extra time required by the operations from our guiding procedure (e.g., code instrumentation time, etc.). All test generations have been performed on the same computer, with a 2.5 GHz Intel i5 processor, 8 GB of RAM, 128 GB of SSD and running Mac OS X 10.8.5. For each run, the JVM has been configured with a heap of 4 GB.

B. Results

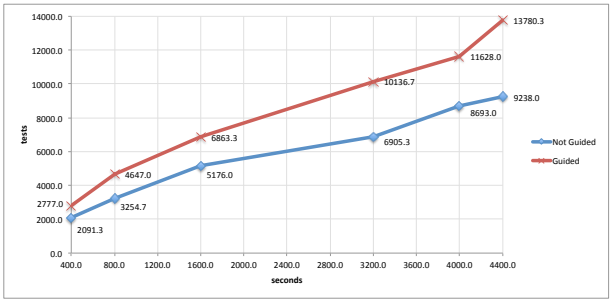
Table I gives the numerical results of our experiments, while Figure 2 shows a visual summary for easier comparison.

[5]http://jung.sourceforge.net/

[6]The original RANDOOP should be deterministic since it uses the same default randomization seed. However, the tested code may introduce non-determinism, the garbage-collector might influence the real execution time, the modified RANDOOP contains some non-deterministic implementation particularities e.g., the "order" of elements when iterating over a set, etc.



(a) Intra-Class Def-Use Pair Coverage



(b) Number of Generated Tests

Fig. 2: Experimental Results

Time Limit (sec.)	Non-Guided Coverage (% avg.)	Guided Coverage (% avg.)	Non-Guided Tests (avg.)	Guided Tests (avg.)
400	25.3	20.1	2 091.3	2 777.0
800	30.8	32.6	3 254.7	4 647.0
1600	35.9	42.5	5 176.0	6 863.3
3200	37.5	45.7	6 905.3	10 136.7
4000	39.7	47.4	8 693.0	11 628.0
4400	40.8	48.5	9 238.0	13 780.3

TABLE I: Experimental Results

As can be observed, for small execution time limits (e.g., 400 seconds), our guiding procedure temporarily achieves lower *intra-class def-use* coverage than the original RANDOOP. However, this is expected: the guiding procedure performs many additional actions (e.g., code instrumentation, determining the *def-use pairs* in a concrete class, capturing runtime information, etc.) especially at the beginning of the test generation process. For a fair comparison, the time allocated to the modified RANDOOP includes these actions and thus its real time limit for test generation is actually smaller. Nevertheless, by inspecting our code during evaluation, we have identified potential ways to speed up these additional tasks and lower their overhead on the test generation procedure.

At about 800 seconds, our guiding procedure starts showing its advantage, see Figure 2(a). For longer time limits, the test suites produced with our guided approach achieve better coverage, by 6.4% on average. For the largest time limits employed, the improvement appears to stabilize at about 8%.

Another observation is that the guiding procedure tends to increase coverage faster that the pure random approach (see the improvement rate between 800 and 1600 seconds in Figure 2(a)). This is consistent with the expected behavior: each time an object of a class containing not yet covered by *intra-class def-use* pairs is detected in a test, the guiding procedure attempts to immediately extend that test by calling the required methods on that object to improve the coverage. Thus, this suggests that the improvements are the results of the guiding procedure itself and that they are not circumstantial.

We should also emphasize that for a time limit greater than 1600 seconds, the coverage improvement rate appears to be similar for the guided and for the unguided RANDOOP. This might be a sign that beyond this threshold the coverage improvement is actually done by the "unguided part" of the modified RANDOOP exposing a limitation of the proposed technique. Intuitively, such a situation may appear when covering an *intra-class def-use* pair requires complex object protocol conditions for invoking the methods containing the targeted *defs/uses* (e.g., not simply invoking a method containing a *use* after a method containing a *def* to the same field).

We also present in Table I and Figure 2(b) the number of tests produced by the modified and unmodified RANDOOP for each time limit, averaged over the three runs. In essence, executing our guiding procedure generates a larger number of tests, by 40% on average. Since more tests are produced in order to increase the coverage with an average of 6.4%, this might be perceived as a disadvantage of our approach. However, this is not necessarily true, because it depends on the actual result revealed by a generated test. For instance, if it exposes an error or an object protocol requirement (e.g., a method must always be invoked before another one) that is not observed by the test, then the additional test is extremely valuable. However, at this time, we have not performed a manual analysis of these tests and thus, we cannot draw a conclusion with respect to this issue.

Evaluating the results for this experiment, we argue that our guiding technique can augment the feedback-directed random test generation in order to more quickly produce better tests with respect to the *intra-class def-use* testing quality criterion. Although the gain might be considered relatively small at this time, we point out that it was obtained with an unoptimized adaptation as discussed at the beginning of this section.

C. Threats to Validity

In terms of external validity, we have used only one system and thus we cannot reliably generalize the conclusion. However, the analyzed program is real (i.e., not fabricated) which should increase the relevance of our results.

In terms of construct validity, our guiding procedure augments the RANDOOP source code. To the best of our understanding, the results should not depend on other interactions besides the intended guiding procedure, however, this cannot completely be excluded.

Another issue regarding construct validity comes from the way in which we approximate *def-use* pairs by identifying them using the methods containing the corresponding statement. However, we consider that this approximation is acceptable for the current state of our work.

V. Related Work

Our paper aims to automatically generate a test suite with good dataflow coverage. Applying dataflow testing to object-oriented programs was first investigated by Harrold and Rothermel [2]. They distinguish between *def-use* pairs at three levels: intra-method, inter-method (when exercised within a call to a single public method), and intra-class (resulting from calls to an arbitrary pair of public methods). We focus on intra-class testing as being the most comprehensive (and needed especially to test libraries).

Finding precise *def-use* relations can be problematic itself; the algorithm presented in [5] takes into account the typical obstacles in object-oriented analysis (dynamic dispatch, imprecise concrete types, aliasing, exceptions) and can be used in test generation. Our prototype uses an approximation by considering method pairs that define and use the same field. A more precise *def-use* analysis would avoid generating tests that try to exercise infeasible *def-use* pairs; at the same time, the reported coverage would be higher, being measured relative to a smaller set of *def-use* pairs to be covered. Tsai et al. [6] show that dataflow anomalies have to be considered in order to obtain a complete set of relevant test cases and do this in a preliminary stage before test case generation.

Several studies provide empirical evidence that dataflow coverage criteria are useful in testing object-oriented software. A test strategy that combines the all-bindings and all du-pairs criteria is presented in [7]. It is shown that more than 80% of object-oriented faults can be detected, although *def-use* relations are tracked only per object rather than at the level of individual fields. In [8], *def-use* coverage is employed also for inter-class (integration) testing, by using contextual information to test state-dependent behavior; this is shown to be effective for detecting seeded mutants.

An approach to detect state-dependent failures is constructed in [9] by augmenting dataflow analysis (which merely produces du-pairs) with symbolic execution to achieve the necessary conditions and automated deduction that generates call sequences conforming to the needed pre- and postconditions. While the combination is supposed to fall back on just dataflow analysis in case of complex programs, it is only illustrated on a simple case study. In comparison, by adapting RANDOOP we have chosen a more light-weight approach, but which is shown to work on a real program of significant size.

To increase coverage, evolutionary algorithms have been employed. Tonella [10] uses genetic algorithms to generate test sequences; relevant features include the methods to invoke and the created objects to use for these invocations, which are also used in our approach to guiding test generation. A genetic algorithm aimed specifically at dataflow testing is presented in [11]; it also achieves a higher coverage than random testing, using a smaller test suite. A large-scale case study, implementing dataflow criteria for the EVOSUITE tool, is reported in [12]. The coverage level achieved (54% of *def-use* pairs) is comparable to our results; despite the relatively low value, achieving a higher mutation score than for branch coverage confirms the benefits of dataflow testing.

VI. Conclusion

We have presented in this paper an approach to guide a well-known random test generation technique in order to more quickly produce better test suites with respect to the *intra-class def-use* coverage testing criterion for object-oriented programs. We have also presented our initial evaluation of the approach emphasizing its potential. Although the identified gain might be considered relatively small, it proves that it deserves investing work in optimizing some implementation details that might improve the current results. Consequently, this would be one of the main directions for future work. At the same time, we would like to identify other test generation guiding approaches to address other object-oriented dataflow testing criteria. Last but not least, we should also investigate the advantages of producing test suites according to the previous criteria in order to find defects within the tested applications.

References

[1] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 75–84.

[2] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *Proceedings, 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. ACM, 1994, pp. 154–163.

[3] INRIA and France Telecom, "Introduction to the ASM 2.0 bytecode framework," http://asm.ow2.org/doc/tutorial-asm-2.0.html.

[4] R. Marinescu, G. Ganea, and I. Verebi, "InCode: Continuous quality assessment and improvement," in *14th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2010, pp. 274–275.

[5] R. Chatterjee and B. G. Ryder, "Data-flow-based testing of object-oriented libraries," Rutgers University, Tech. Rep. DCS-TR-433, 2001.

[6] B.-Y. Tsai, S. Stobart, and N. Parrington, "Employing data flow testing on object-oriented classes," *IEE Proceedings - Software*, vol. 148, no. 2, pp. 56–64, 2001.

[7] M.-H. Chen and H. Kao, "Testing object-oriented programs - an integrated approach," in *10th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 1999, pp. 73–82.

[8] G. Denaro, A. Gorla, and M. Pezzè, "An empirical evaluation of data flow testing of Java classes," University of Lugano, Tech. Rep. 2007/03.

[9] U. A. Buy, A. Orso, and M. Pezzè, "Automated testing of classes," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2000, pp. 39–48.

[10] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2004, pp. 119–128.

[11] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using genetic algorithms to aid test-data generation for data-flow coverage," in *14th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2007, pp. 41–48.

[12] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 370–379.