Test Generation across Multiple Layers

Matthias Höschele · Juan Pablo Galeotti · Andreas Zeller
Saarland University, Saarbrücken, Germany
{hoeschele, galeotti, zeller}@cs.uni-saarland.de

ABSTRACT

Complex software systems frequently come in many layers, each realized in a different programming language. This is a challenge for test generation, as the semantics of each layer have to be determined and integrated. An automatic test generator for Java, for instance, is typically unable to deal with the internals of lower level code (such as C-code), which results in lower coverage and fewer test cases of interest.

In this paper, we sketch a novel approach to help search-based test generators for Java to achieve better coverage of underlying native code layers. The key idea is to apply test generation to the native layer first, and then to use the inputs to the native test cases as targets for search-based testing of the higher Java layers. We demonstrate our approach on a case study combining KLEE and EvoSuite.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Reliability, Theory

Keywords

Search based software engineering, symbolic execution, unit testing, test generation, multiple layers, native code

1. INTRODUCTION

Testing is an important task in the development of software systems. Tools for automatically generating test cases aim at lowering the cost of creating tests by removing part of this burden from the user. Search-Based Software Testing [13] (SBST) cast the problem of creating a test suite as an optimization problem. In the context of SBST for Java programs, one recurrent limitation lays in dealing with native code. The Java Native Interface (JNI) [11] allows Java code running in a Java Virtual Machine to call applications and libraries written in lower-level languages such as C, C++ and assembly. In contrast to the Java bytecode, this native code is specific to the hardware and operating system. Java developers choose native implementations to increase code reusability (re-using already implemented and debugged C/C++ or assembly code) and performance (by avoiding unnecessary layers of abstraction between the machine execution and the program). Native methods are declared in a Java class by simply adding the `native` method modifier.

Every Java program involving native function calls could be seen as a layered system where the Java code represents the *upper* layer (since all the native functions are invoked through the Java code) and the native module could be seen as the *lower* layer. Test case generation for such systems is challenging. Apart from the obvious implementation problems caused by dealing with heterogeneous environments, techniques that are suitable for object-oriented programs might not apply to low-level procedural languages and vice versa. An SBST tool such as EvoSuite [3] might be able to efficiently deal with the object-oriented aspects of the upper layer (like synthesizing sequences of method invocations to create complex objects) and achieve high coverage, but since native code is ignored, there is no guidance to improve the search towards uncovered goals. On the other hand, Dynamic Symbolic Execution [9, 14] (DSE) has been particularly successful at dealing with the caveats of low-level code. In particular, KLEE [2] arises as a very effective tool for achieving high coverage of low-level and system code, although applying KLEE on the low-level code in isolation might lead to scenarios that are practically infeasible considering the high level framework.

In the context of layered-systems, we intend to find a sweet spot where SBST tools could improve coverage and generate more meaningful test cases by means of applying symbolic execution tools at the native code level. Figure 1 shows a sketch of our technique for generating test cases for layered systems:

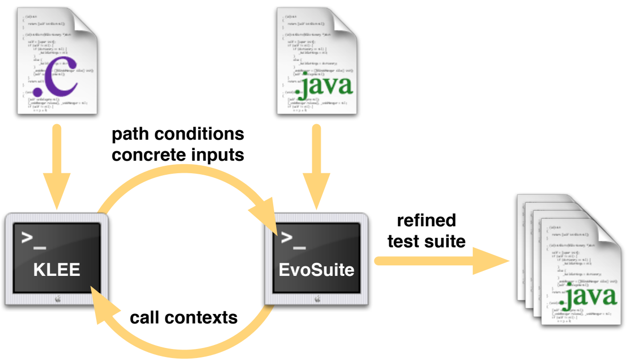

Figure 1: Overview of our proposed approach

SBST'14, June 2 – June 3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2852-4/14/06...$15.00
http://dx.doi.org/10.1145/2593833.2593834

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y)
    if (f(x) == y + 1000000)
      return -1;
  return 0;
}
```

Figure 2: C code of the `foo` native module.

1. First, we *systematically explore the lower-layer native C modules* with a symbolic execution tool such as KLEE. The symbolic tool builds concrete inputs to reach high structural coverage, as well as path conditions (i.e. constraints) on the inputs to pursuit a particular path on the native code.
2. Then, we *instrument* the Java upper-layer with *additional branches* that encode the concrete inputs found as well as the collected path conditions from symbolic execution.
3. We apply a SBST tool such as EvoSuite on the instrumented code. The search is *influenced by the additional branches,* forcing the execution of the native code to follow a particular path in the low-level layer.
4. By collecting the concrete native invocations from the test suite (i.e. array sizes, field values), we can *spawn new symbolic explorations* from each interesting calling context. These then lead to previously unknown concrete inputs and path conditions, which are again fed to the SBST tool.

This process continues until all goals have been covered (both at the upper as well as the lower layers) or the test generation budget has been exhausted.

2. CROSS-LAYER TEST GENERATION

Let us illustrate our approach by an example. Figure 2 shows a native module with functions `h()` and `f()` written in C (a slightly modified version of the motivating example presented in [9]). Observe that, just by simply exercising random values of `x` and `y` it is highly unlikely that `h()` returns a value different than `0`. Dynamic Symbolic Execution [9] tackles this particular problem by gathering constraints from the program execution, which are later negated and fed to a constraint solver. Solutions to this constraint systems are then translated into test inputs in order to achieve higher structural coverage.

Figure 3 introduces a Java class invoking the native library `foo` which contains the C functions in Figure 2. This code represents an *upper* layer in the complete system, since all the native C functions are invoked from the Java class.

Any unit test case generator for Java will find executing the native function `h()` declared in `Foo` quite challenging. First, the visibility level is private. The only way a test case generator can exercise this function is by means of its caller: method `bar`. Secondly, since the behavior of `bar()` depends on object state, the tool will have to create sequences of method invocations to properly *"start"* the `Foo` instance before invoking `bar()` on it.

EvoSuite's genetic algorithm evolves entire test suites towards structural coverage criteria. If we apply EvoSuite to automatically generate a test suite for the `Foo` class, it is highly likely that the resulting test cases will not exhibit the `AssertionError`. This is due to the fact that the search will have no guidance to cover the branch triggering the problem.

On the contrary, if we consider our proposed approach on this particular example, we start by applying KLEE on the C-code

```
class Foo {
  // make h() available as a native method
  static {
      System.loadLibrary("foo");
  }
  private static native int h(int x, int y);

  // Java code
  private boolean flag = false;
  void start() {flag = true;}
  void stop()  {flag = false;}
  void bar(int value1, int value2) {
    if (flag) {
      int retValue = h(value1,value2);
      if (retValue != 0) {
         // a failure has occurred
        throw new AssertionError();
       }
    }
  }
}
```

Figure 3: A Java class with a declared native function `h()` in the `Foo` native module

```
int main() {
 int x;
 int y;
 klee_make_symbolic(&x, sizeof(x), "x");
 klee_make_symbolic(&y, sizeof(y), "y");
 return h(x,y);
}
```

Figure 4: Execution harness for executing KLEE on the function `h()`

shown in Figure 2. In order to do so, we create the execution harness shown in Figure 4. One important aspect of creating this execution harness is deciding what should be symbolic (some integer values) and what should be concrete (i.e. array sizes). KLEE's symbolic execution will cover all paths with exactly 3 values for `x` and `y`, shown in Table 1. In particular, invoking `h()` with values `x==1056948234` and `y==2112896468` returns -1. Then, we instrument the original Java source code as follows:

1. We add a new method `instr_h()` declared in Figure 5. An `if` condition is inserted for each of the concrete values found by KLEE. In turn this method invokes the native function `h()`.
2. Additionally, each path condition collected by KLEE is encoded as a branch decision and inserted to `instr_h()`.
3. Each invocation to `h()` in the Java class is replaced by an invocation to `instr_h()`.

The instrumented Java class `Foo` is equivalent in behaviour (since all conditions are side-effect free), but the appended branches provides twofold guidance to EvoSuite: One is provided by concrete inputs found by KLEE that can easily be picked up by EvoSuite

Table 1: Parameters and return values for native function `h()` generated by KLEE

x	y	h(x,y)
0	0	0
0	1	0
1056948234	2112896468	-1

```
private static int instr_h(int x, int y) {
  /* concrete inputs found by KLEE */
  if (x == 1056948234 && y == 2112896468) {
       /*do nothing*/
  }
  if (x == 0 && y == 0) {
       /*do nothing*/
  }
  if (x == 0 && y == 1) {
       /*do nothing*/
  }
  /* path conditions collected by KLEE */
  if (x==y) {
       /*do nothing*/
  }
  if (x!=y && 2*x!=y+1000000) {
       /*do nothing*/
  }
  if (x!=y && 2*x==y+1000000) {
       /*do nothing*/
  }
  /* invocation to native method h() */
  return h(x,y);
}
```

Figure 5: The instrumented version of the `h()` native function from Figure 3. The first conditions encourage the use of values provided by KLEE

```
@Test
public void test6() throws Throwable  {
    Foo foo = new Foo();
    foo.start();
    // Undeclared exception!
    try {
      foo.bar(1056948234, 2112896468);
      fail("Expecting exception AssertionError");
    } catch(AssertionError e) {
    }
}
```

Figure 6: A JUNIT **test case generated automatically by** EVOSUITE **on the instrumented code of Figure 5**

as suggestions for values[1]. But it could be the case that one of these concrete solutions found during symbolic exploration is infeasible in the context of the more limited usage within the Java class. In order to address this issue, we also instrument the path conditions. By doing this, if a concrete solution is infeasible, but there is an alternative solution for the same path, EVOSUITE might be guided by the path condition branches.

The execution of EVOSUITE on the instrumented Java bytecode leads to the test case presented in Figure 6. *The resulting test suite achieves full coverage and exhibits the failure in the code.* A programmer inspecting the resulting test suite will conclude the existence of a problem since the reflected behavior was not expected.

By sharing constraints and calling contexts between KLEE *and* EVOSUITE*, we may be able to generate interesting tests for systems across a high-level Java layer and a low-level C layer*.

We presume that applying different techniques targeting specific features of each layer might pave the way towards generating tests cases for systems with multiple layers.

3. THE ROAD AHEAD

While the initial example works well in practice, our concept for testing across layers is still under development. Issues we currently face include:

- Providing feedback from EVOSUITE to KLEE requires the recording of call contexts. To do this we need to extend EVOSUITE or add further instrumentation to the Java bytecode that records array sizes and serializes objects to obtain field values.
- When combining multiple tools into one approach, finding the right allocation of resources is crucial to achieve good results. We expect the optimal ratio of time spent on SBST and symbolic execution to be dependent on the idiosyncrasies of subjects and will provide it as a configurable option.
- JNI provides access to Java heap memory via API calls to the virtual machine. This complex interaction might be hard to handle for KLEE. We need to replace these calls during symbolic execution in order to provide the runtime information (i.e. object field values) recorded while running EVOSUITE.
- Callbacks from C code to Java methods can also be a challenge since these methods are out of reach for a tool like KLEE. We need to investigate how frequently such callbacks are used and if their use poses a threat to the applicability of our approach.
- There are small semantic differences regarding primitive types in Java and C. When transferring path conditions or concrete values between those environments we need to be aware of these differences and possibly modify values in order to preserve the intended semantics.

Our approach is not limited to the simple stack of Java code and native C methods, but can also be applied to more complex stacks like Web applications (Javascript / PHP / SQL / C) or applications interacting with operating system kernels. Another application is to stay within one language, and using symbolic execution for as many lower layers as possible, leaving it to SBST to cover the higher layers and/or system interaction. Despite their challenges, we see great potential for cross-layer test generation—making test generation scale up not only to systems, but to systems of systems.

4. RELATED WORK

4.1 Search-based Testing

The field of search-based software testing (SBST) is concerned with the application of efficient search algorithms in order to generate test cases. *Genetic Algorithms* (GA) are one of the most commonly applied classes of global search algorithms. The intuition behind GAs is to imitate the natural process of evolution. Populations of candidate solutions are evolved by probabilistically applying domain specific operators that mimic mutation and recombination via crossover. The parents for individuals of the next generation are chosen based on their fitness in order to bias the survival of the fittest. The GA continually improves the fitness of the population until either an optimal solution is reached or a stopping criterion is met (e.g. maximum number of fitness evaluations or time limit). In the domain of evolutionary testing, a population would represent a set of test cases and the fitness measures how close a candidate solution is to satisfy a coverage goal.

[1]EVOSUITE extracts constants from the byte code in order to seed values during the search, which has tremendous impact in the search [4]

Fitness functions are used to increase the likelihood of choosing promising individuals for reproduction and thereby gradually improving the fitness of each generation. In evolutionary testing fitness functions for branch coverage [13] usually integrate the *approach level* (number of unsatisfied control dependencies) and the *branch distance* (heuristic for how close the deviating condition is to evaluating as desired). Such search techniques are not restricted to the context of primitive datatypes, but have also been applied to test object-oriented software using method sequences [15, 6].

The conventional approach to SBST is to search for test cases that satisfy a single coverage goal in isolation. Since limited computational resources need to be allocated to the entire set of coverage goals, dead code and unsatisfiable branch conditions could waste a lot of runtime that could help to cover feasible branches. Whole test suite generation implemented by EvoSuite [5] tries to fix this issue by optimizing an entire test suite at once towards satisfying a coverage criterion, and therefore prevents results from being adversely effected by the order, the difficulty or infeasibility of individual coverage goals.

4.2 Symbolic Execution

KLEE [2] is a symbolic execution tool. In order to execute symbolically, the user has to provide an execution harness as the one presented in Figure 4. In KLEE, programs are compiled into LLVM assembly, a reduced virtual instruction set. Every time a LLVM instruction is symbolically executed, the current symbolic state is updated. When KLEE hits a branch instruction, the branch condition is symbolically evaluated to true and false. Then, a new constraint system is built by conjoining each of these new conditions to the current path condition associated to this symbolic state. Then, each constraint system is later fed into the STP [8] constraint solver. If neither of them is satisfiable, the current symbolic state is discarded. If both are satisfiable, the symbolic state is cloned, and the each path condition is updated with the corresponding branch condition. If only one branch is satisfiable, no state cloning happens, although the satisfiable branch condition is appended to the current path condition. This process continues until no more symbolic states are available for exploration, or the exploration budget has been exhausted.

When the exploration is finished, KLEE summarises the concrete input values of each variable for each satisfiable path. KLEE also models some memory and arithmetic errors (such as division by zero, null dereferences, buffer overflows, accessing invalid memory addresses, etc.) as new branches. By doing so, KLEE provides the user with an input exhibiting potentially harmful behaviour.

4.3 Hybrid Approaches

There are several approaches to combine SBST and symbolic execution. In [12], DSE is introduced as a mutation operator to escape local optima during the search-based algorithm. In [7, 10] the genetic algorithm is periodically *alternated* with DSE. Current chromosomes are transformed in parametrised unit tests in order to apply DSE. On the other hand, new executions obtained by DSE are later transformed into chromosomes that are fed to the evolutionary approach. Baars et al. [1] introduced the idea of symbolic search-based testing, where the fitness function of the GA takes different possible symbolic paths to the target into account. All these techniques share the goal of achieving maximum structural coverage by means of intertwining SBST and DSE. In contrast, in the context of the current work, these SBST and symbolic execution are used in different domains.

5. CONCLUSIONS

With the concept of cross-layer test generation, we have sketched a high level overview on how to combine a symbolic execution tool like KLEE and a SBST-based tool like EvoSuite in order to achieve better coverage in layered systems. We also presented the main challenges we might like to address in the near future. In the long run, cross-layer test generation may prove as a promising approach to scale test generation from individual libraries and applications to complex systems which no single test generation technique could handle as a whole.

Acknowledgments.

This work was funded by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing". Eva May provided helpful comments on earlier revisions of this work.

6. REFERENCES

[1] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos. Symbolic search-based testing. In *ASE*, pages 53–62, 2011.

[2] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[3] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *FSE*, pages 416–419, 2011.

[4] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *ICST*, pages 121–130. IEEE, 2012.

[5] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[6] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *TSE*, 28(2):278–292, 2012.

[7] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *ISSRE*, pages 360–369. IEEE, 2013.

[8] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[10] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, pages 297–306. IEEE, 2008.

[11] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[12] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *ASE*, pages 436–439, 2011.

[13] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[14] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[15] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, 2004.