Palus: A Hybrid Automated Test Generation Tool for Java

Sai Zhang
Department of Computer Science & Engineering
University of Washington
szhang@cs.washington.edu

ABSTRACT

In object-oriented programs, a unit test often consists of a sequence of method calls that create and mutate objects. It is challenging to automatically generate sequences that are *legal* and *behaviorally-diverse*, that is, reaching as many different program states as possible.

This paper proposes a combined static and dynamic test generation approach to address these problems, for code without a formal specification. Our approach first uses dynamic analysis to infer a call sequence model from a sample execution, then uses static analysis to identify method dependence relations based on the fields they may read or write. Finally, both the dynamically-inferred model (which tends to be accurate but incomplete) and the statically-identified dependence information (which tends to be conservative) guide a random test generator to create legal and behaviorally-diverse tests.

Our Palus tool implements this approach. We compared it with a pure random approach, a dynamic-random approach (without a static phase), and a static-random approach (without a dynamic phase) on six popular open-source Java programs. Tests generated by Palus achieved 35% higher structural coverage on average. Palus is also internally used in Google, and has found 22 new bugs in four well-tested products.

Categories and Subject Descriptors

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

General Terms

Reliability, Experimentation

Keywords

Automated test generation, static and dynamic analyses

1. INTRODUCTION

In an object-oriented language like Java, a good unit test requires desirable method call sequences (in short, *sequences*) that create and mutate objects. These sequences help generate target object states of the receiver or arguments of the method under test.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

To alleviate the burden of writing unit tests manually, many automated test generation techniques have been studied [1–3, 10, 12]. Of existing test generation techniques, bounded-exhaustive [2, 10], symbolic execution-based [7, 15], and random [3, 9, 12] approaches represent the state of the art. Bounded-exhaustive approaches generate sequences exhaustively up to a small bound on sequence length. However, generating test to reach many program states often requires longer sequences beyond the small bound that could be handled. Symbolic execution-based tools like JPF [13] and CUTE [15] explore paths in the tested program symbolically and collect symbolic constraints at all branching points of an explored path. The collected constraints are solved if feasible, and a solution is used to generate an input for a specific method. However, these symbolic execution-based tools face the challenge of scalability, and do not provide effective support for generating method sequences. Furthermore, the quality of generated inputs heavily depends on the test driver provided (which is often manually written [7, 13, 15]). Random approaches [1, 3, 12] have been demonstrated to be easy-to-use, scalable and able to find previously-unknown bugs [11], but they face challenges in achieving high structural coverage for certain programs. One major reason is that, for programs with constrained interfaces, correct operations require calls to occur in a certain order with specific arguments. Thus, most randomly-created sequences could be illegal, or may fail to reach new target states.

This paper presents a technique (and its tool implementation called Palus) that improves random test generation by incorporating results obtained from a dynamic analysis and a static analysis. Static and dynamic analysis are two mainstream program analysis techniques. Static analysis examines program code and reasons over all possible behaviors that might arise at run time, while dynamic analysis operates by executing a program and observing the executions. They could enhance one another by providing information that would otherwise be unavailable. The need to combine the two approaches has been repeatedly stated in the literature [4, 6]. Our hybrid approach takes a correct sample execution as input, infers a call sequence model [1], and enhances it with argument constraints. This enhanced call sequence model captures *legal* method call orders and argument values observed from the sample execution. Then, our approach uses a static analysis to explore the possible dependence relations of methods under test. The static analysis includes a tf-idf-based weighting scheme [8] for ranking the dependence relevance between two methods and a method dependence relevance measurement that reflects how tightly two methods are coupled. In general, methods that read and write the same field are dependent. Testing them together has a higher chance of exploring different program behaviors. Finally, both the dynamically-inferred model and the statically-identified dependence information guide a random test generator [12] to create legal and behaviorally-diverse

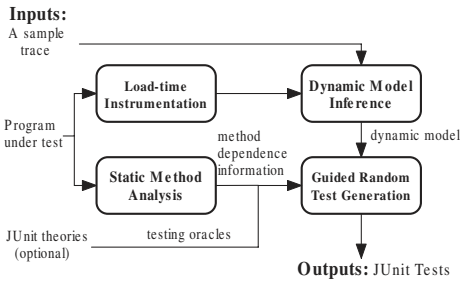

Figure 1: Architecture of the Palus tool.

tests. Thus, our combination has three steps: dynamic inference, static analysis, and *guided* random test generation.

Several past research tools follow an approach similar to ours, but omit one or two of the three stages of our approach. Randoop [12] is a pure random test generation tool. Palulu [1] is a representative of the dynamic-random approaches: it infers a call sequence model from a sample execution, and follows that model to create tests. However, the Palulu model lacks necessary constraints for method arguments, and has no static analysis phase to enrich the dynamically-inferred model and could thus miss execution-uncovered methods in creating tests. Finally, RecGen [17] uses a static-random approach. RecGen does not have a dynamic phase, and uses a heuristic-based static analysis to guide random test generation. Since RecGen lacks the guidance from an accurate dynamic analysis, it may fail to create legal sequences for programs with constrained interfaces, and then miss target states.

Unlike past research tools [1, 12, 17] that only check generalized programming rules (e.g., for each Java object `o`: `o.equals(o)` returns true), Palus seamlessly integrates with the JUnit theory framework [14], permitting programmers to write project-specific, comprehensive testing oracles.

Our approach is novel in how it uses information from a dynamic analysis to create legal sequences and uses information from a static analysis to diversify the generated sequences. Thus, our approach could be regarded as *fuzzing on a specific legal path*. In our evaluation (Section 3), tests generated by Palus achieved much higher structural coverage, and detected bugs in real-world software.

2. APPROACH

Figure 1 gives an overview of our approach. Palus takes Java bytecode as input and performs load-time instrumentation. It collects traces from the sample execution and infers an enhanced call sequence model for each class under test. Palus also analyzes the program under test statically, and computes the set of fields that may be read or written by each method. Using field accessing (read and write) information, a pair-wise method dependence relevance value will be calculated to reflect how closely two methods are coupled. Finally, the call sequence model and the method dependence information guide the sequence generation process.

2.1 Dynamic Analysis: Model Inference

We design an enhanced call sequence model to capture the possible legal method call sequences and argument values.

A call sequence model [1] is a rooted, directed, and acyclic graph. Each model is constructed for one class observed during execution. Edges (or *transitions*) represent method calls and their arguments, and each node in the graph represents a collection of object states, each of which may be obtained by executing the method calls along some path from the root node. Each path starting at the root corresponds to a legal sequence of calls that operate on a specific object — the first method constructs the object, while the rest of them mutate the object (possibly as one of method parameters).

Palus enhances the call sequence model with two kinds of argument constraints, namely *direct state transition dependence constraint* and *abstract object profile constraint*. The direct state transition dependence constraint is represented by an edge from one state (source node) to another (destination node) in call sequence models. Such a dependence edge represents a possible argument object state. That is, an object state at the source node may be used as an argument when extending a model sequence at the destination node. The abstract object profile constraint uses an abstraction function [5] to map each captured argument value to an abstract value, and keeps them in the model. Palus classifies all created object instances, and selects needed ones based on their abstract state profiles when generating sequences. Such profiles, though coarse, enable the selection of desirable objects as arguments.

2.2 Static Analysis: Model Expansion

The dynamically-inferred model provides a good reference in creating legal sequences, but it is incomplete, and may miss covering some interesting methods or certain method call invocation orders. To alleviate this problem, we devise a lightweight static analysis to enrich the model. Our static method dependence analysis hinges on the hypothesis that two methods are related if the fields they read or write overlap. Testing two related methods has a higher chance of exploring new program states. Thus, when creating a test, Palus prefers to invoke related methods together.

Palus computes two types of dependence relations: (1) *write-read* relation: if method `f` reads field `x` and method `g` writes field `x`, we say method `f` has a *write-read* dependence relation on `g`; and (2) *read-read* relation: if two methods `f` and `g` both read the same field `x`, each has a *read-read* dependence relation on the other.

Palus first computes the read/write field set for each method, then merges the effects of method calls. One method may depend on multiple other methods. We devise a tf-idf (term frequency–inverse document frequency)-based weighting scheme [8] to measure how tightly each pair of methods is coupled. Afterwards, in the sequence generation phase, when a method `f` is tested, its most dependent methods are most likely to be invoked after it.

2.3 Guided Random Test Generation

We design a guided random test generation algorithm using both the dynamically-inferred model and the statically-identified dependence information.

The algorithm works in two phases, each of which consumes a fraction of the given time limit. In the first phase, Palus repeatedly performs one of the following actions: (1) create a new sequence from an enhanced call sequence model root, and (2) extend an existing sequence along a randomly-selected model transition. When extending an existing sequence, Palus incorporates the statically-identified dependence information to invoke one dependent method together, and concatenates it to create a new sequence. In the second phase, to avoid missing model-uncovered methods, Palus randomly creates [12] sequences for those model-uncovered methods. Every created sequence is checked against a set of generalized properties [12] as well as user-provided oracles (Section 2.3.1). Any violating tests are classified as failure-revealing tests.

2.3.1 Oracle Checking

Besides checking generalized Java programming rules (e.g., the symmetry property of equality: `o.equals(o)`), Palus integrates with the JUnit theory framework [14] (which is similar to Parameterized Unit Tests [16]), permitting programmers to write domain-specific oracles.

```
@Theory
void noExceptionInHasNext(Iterator iterator){
    Assume.assumeNotNull(iterator);
    try {
        iterator.hasNext();
    } catch (Exception e) {
        fail("hasNext() should never throw an exception!");
    }
}
```

Figure 2: A testing oracle expressed in the JUnit theory form. It checks for any non-null `Iterator` object, no exception should be thrown when invoking its `hasNext()` method.

Program	LOC	Line Coverage%				Inc%
		Randoop	Palulu	RecGen	Palus	
tinySQL	7672	29	41	29	**57**	**72%**
SAT4J	9565	44	44	-	**65**	**47%**
JSAP	4890	62	70	68	**72**	**8%**
Rhino	43584	22	25	25	**28**	**14%**
BCEL	24465	36	39	29	**53**	**53%**
Apache	55400	38	29	28	**38**	**19%**
Average	24262	38	41	36	**52**	**35%**

Table 1: Experimental results of comparison between Randoop (a pure random approach), Palulu (a dynamic-random approach), RecGen (a static-random approach) and Palus. Column "LOC" shows the lines of code of each program. Column "Inc%" shows the average coverage improvement achieved by Palus. RecGen fails to generate compilable tests for SAT4J.

A theory is a generalized assertion that should be true for any data. Take a sample theory in Figure 2 as an example, Palus will automatically check for every non-null `Iterator` object, that the assertion should hold. When Palus executes a theory with concrete object values, if an `Assume.assume*` call fails and throws an `AssumptionViolatedException`, Palus will intercepts this exception and silently proceeds. If some generated inputs cause an `Assert.assert*` to fail, or an exception to be thrown, the failures are outputted to the programmers.

3. EVALUATION

We compared the structural coverage achieved by Palus and three existing tools in Table 1. Overall, Palus achieved 35% higher coverage on average.

A slightly modified version of Palus is internally used at Google. We evaluated it on four large-scale products, including a testing server, Google Buzz, AdWords, and Orkut. The source code of each product is required to be peer-reviewed, and goes through a rigorous testing process before being checked into the code base. Each product has a comprehensive unit test suite that achieves line coverage ranging from 57% to 72%. As a result, Palus revealed 22 previously-unknown bugs in the four products. Each bug has been submitted with the generated test, and some of them have since been fixed. The bugs found by Palus eluded previous testing and peer review. The primary reason we identified is that for large-scale software of high complexity, testers often miss corner cases. While Palus makes no guarantees about covering all cases, its hybrid test generation strategy permits it to create legal tests to discover many missed corner cases for which no manual tests are covering. This suggests Palus can be effective in finding real-world bugs in large, well-tested products.

The source code of Palus and the experimental data on open-source programs are publicly available at:

`http://code.google.com/p/tpalus/`

Acknowledgment. We thank Michael Ernst, David Saff, Yingyi Bu, Todd Shiller, Nicolas Hunt, and the anonymous reviewers for their feedback. Part of the experiment was done during a summer internship in Google. This work was supported in part by ABB Corporation and NSF under grant CCF-0963757.

4. REFERENCES

[1] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Portland, OR, October 23, 2006.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proc. ISSTA '02*, pages 123–133, 2002.

[3] C.Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. In *Software: Practice and Experience, 34(11)*, pages 1025–1050, 2004.

[4] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–37, 2008.

[5] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10*, pages 85–96, 2010.

[6] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, May 9, 2003.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc.PLDI '05*, pages 213–223, New York, NY, USA, 2005. ACM.

[8] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[9] K.Claessen and J.Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ICFP '00*, pages 268–279, Sept.2000.

[10] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of java programs. In *Proc. ASE '01*, page 22, 2001.

[11] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *Proc. ISSTA 2008*, July 20–24, 2008.

[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84, Minneapolis, MN, USA, 2007.

[13] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA '08*, pages 15–26, New York, NY, USA, 2008. ACM.

[14] D. Saff, M. Boshernitsan, and M. D. Ernst. Theories in practice: Easy-to-write specifications that catch bugs. Technical Report MIT-CSAIL-TR-2008-002, MIT CSAIL, Cambridge, MA, January 14, 2008.

[15] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE-13*, pages 263–272, 2005.

[16] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.

[17] W. Zheng, Q. Zhang, M. Lyu, and T. Xie. Random unit-test generation with MUT-aware sequence recommendation. In *Proc. ASE 2010*, September 2010.