# Combined Static and Dynamic Automated Test Generation

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. These sequences help generate target object states for the program under test. Automatic generation of good sequences is often challenging, because: (1) there could be many constraints in creating *legal* sequences, and (2) sequences that achieve high structural coverage or reveal bugs need to be *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 this problem without a formal specification. Our approach first uses dynamic analysis to infer an enhanced 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 dynamically-inferred model (that tends to be accurate but incomplete) and statically-identified dependence information (that tends to be conservative) guide a random test generator to create legal and behaviorally-diverse tests.

Our Palus tool implements this approach. We assessed its effectiveness on six popular open-source Java programs. We compared its effectiveness with a pure random approach, a dynamic-random approach (without a static phase), and a static-random approach (without a dynamic phase). We showed that tests generated by Palus achieve 35% higher structural coverage on average than existing approaches. Palus was also internally used in Google, and found 22 new bugs in four well-tested commercial products.

## 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.

To alleviate the burden of writing unit tests manually, many automated test generation techniques have been studied [2,5,6,15,20, 23,29]. Of existing test generation techniques, bounded-exhaustive [4, 18], symbolic execution-based [11,28], and random [5, 15, 23] approaches represent the state of the art. Bounded-exhaustive approaches generate sequences exhaustively up to a small bound of sequence length. However, generating test to reach many program states often requires longer sequences beyond the small bound that could be handed. Symbolic execution-based tools like JPF [24] and CUTE [28] explore paths in program being tested 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 depend on the test driver provided (which is often manually written [11, 24, 28]). Random test generation approaches (and its variants [20, 29]) have been demonstrated to be easy-to-use, scalable and able to find previously-unknown bugs [22, 23], but they face challenges in achieving high structural coverage for certain programs. One major reason is that, for programs that have constrained interfaces, correct operations require calls occur in a certain order with specific arguments. Thus, most randomly-created sequences could be illegal; and have a low probability of reaching many target states.

This paper presents an automated technique to generate unit tests for object-oriented programs by incorporating results obtained from static and dynamic analyses. Static and dynamic analysis are two mainstream program analysis techniques in the software engineering community. 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 [8, 9, 33]. In this work, our hybrid approach takes a correct execution as input, infers a call sequence model [2], and enhances it with argument constraints. This enhanced call sequence model captures *legal* method-call orders and argument requirement observed from the sample execution. Then, our approach employs a static analysis to explore the possible dependence relations of methods under test. The static analysis includes a variant of the tf-idf weighting scheme [13] for ranking the dependence relevance between two methods and a method dependence relevance measurement that reflects how tightly the 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 dynamically-inferred model and statically-identified dependence information guide a random test generation technique [23] to create legal and behaviorally-diverse 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. Clearly, Randoop [23] is a pure random test generation tool. Palulu [2] 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, a static-random approach is implemented by Zheng et al [35] in their RecGen tool. RecGen lacks the guidance from an accurate dynamic analysis, and could fail to create legal sequences for programs that have many constraints, missing many target states.

Unlike past research tools [2, 23, 35] that only check generalized programming rules (e.g. for each Java object: `o.equals(o)` returns true), our tool implementation Palus seamlessly integrates with the JUnit theory framework [26]. This permits programmers to write project-specific, comprehensive testing oracles.

The novelty of our approach over existing ones is to use information from dynamic analysis to guide the creation of legal sequences, and use information from static analysis to diversify generated sequence. Thus, our approach could be regarded as *fuzzing along a specific legal path*. As shown in Section 4, our approach achieves a considerable improvement in test coverage and finds real bugs.

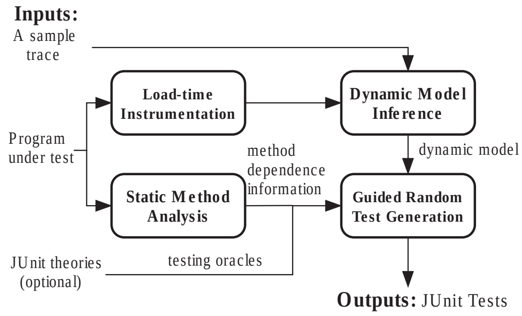


**Figure 1: Architecture of the Palus tool.**

## 2. RELATED WORK

Work related to our approach falls into two main categories; (1) techniques for inferring program behavior models, and (2) tools for automatically generating unit tests.

### 2.1 Program Behavior Model Inference

A rich body of work has been done on program behavior model (or specification) inference from either source code [32] or dynamic executions [7, 10, 17]. The concept of learning models from actual program runs was pioneered in [7] by applying a probabilistic NFA learner on C traces. Their approach relies on manual annotations to relate functions to objects. Dynamic invariant detection techniques [10] express properties of data that hold at specific moments during the observed executions. The work by Whaley et al. [32] mines models with anonymous states and slices models by grouping methods that access the same fields. Later, Lorenzoli et al. [17] mined extended finite state machines with anonymous states, and used their GK-Tail algorithm to merge states and compress models. Compared to other specification mining techniques, the dynamically-inferred model used in this paper is designed for test generation. It gives guidance in creating a legal sequence that reproduces observed behavior, but does not try to generalize the observations (e.g., inferring temporal properties such as that method `f` is always called after method `g`).

### 2.2 Automated Test Generation

Many automated test generation techniques for object-oriented programs [5, 6, 21, 23] have been proposed in the last decade. For example, JCrasher [5] creates test inputs by using a parameter graph to find method calls whose return values can serve as input parameters. Eclat [21] and Randoop [23] use feedback information as guidance to generate random method-call sequences. AutoTest [6] uses a formal specification provided by programmers to check whether randomly-generated tests are error-revealing. However, the sequence creation phase of all the above work is random in nature, thus it can be difficult for these tools to generate good sequences for programs with constrained interfaces.

To handle the large search space of possible sequences, data mining techniques, such as frequent itemset mining, have been adapted to improve the effectiveness of test generation. MSeqGen [29] mines client code bases statically and extracts frequent patterns as implicit programming rules that are used to assist in generating tests. Such approaches can be sensitive to the specific client code, and thus the results may heavily depend on the quality of client code base provided by the code search engine. To avoid this problem, the static analysis phase in our approach takes a different perspective. It emphasizes how methods are implemented rather than how they are used, which is insensitive to a specific client code. It is based on the observation that in general methods are related because they share state (e.g., the fields they read or write). Such coupled methods should be tested together to increase the chance to reach more program states.

Another two alternative approaches to create object-oriented unit tests are via direct heap manipulation and using capture and replay techniques. Korat [4] and TestEra [18] are two representative techniques for direct object construction. They require users either to provide a `repOK()` predicate method or to specify class invariants. In contrast, our approach does not require a manually-written invariant to create tests. Instead, our approach infers a model and uses static analysis to guide the random search towards legal and diverse sequences. On the other hand, the Object Capture-based Automated Test (OCAT) approach [12] uses capture and replay techniques to save object instances from sample executions, and then reuses these saved object instances (serialized on disk) as parameter values when creating sequences. Compared to this work, OCAT does not create a sequence to generate the saved object instance, so OCAT might be less useful for programmers who wish to understand how the object instance can be created. Besides, OCAT is designed as a regression test generation technique, and does not achieve the objective of bug finding (in either their methodology or experimental results).

## 3. 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 then builds 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 inferred call sequence model and method dependence information guide the feedback-directed random test generation [23].

### 3.1 Dynamic Analysis: Model Inference

We devised an enhanced call sequence model to capture the possible legal method-call sequences and arguments. Palus will infer such a model from a given sample trace.

#### *3.1.1 Call Sequence Model Inference*

A call sequence model [2] is a rooted, directed, 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 sequence of calls that operate on a specific object — the first method constructs the object, while the rest mutate the object (possibly as one of their parameters).

The construction of the call sequence model is object-sensitive. That is, the construction algorithm first constructs a call sequence graph for each object of the class observed during execution. Then, it merges all call sequence graphs of objects of the class. Thus, the final call sequence model is a summary of the call sequence graphs for all instances of the class. The call sequence model handles many Java features like nested calls, recursion, and private calls. The detailed definition, construction steps, and inference algorithms appear in [2].

#### 3.1.2 Model Enhancement with Argument Constraints

The call sequence model is too permissive: using it can lead to creating many sequences that are all illegal and thus have similar, uninteresting behavior. Our approach enhances a call sequence model with two kinds of method argument constraints. *Direct state transition dependence constraints* are related to how a method argument was created. *Abstract object profile constraints* are related to the value of a method argument's fields.

***Direct State Transition Dependence Constraint.*** This constraint represents the state of a possible argument. An edge from node A to B indicates that an object state at A may be used as an argument when extending a model sequence at B.

A state dependence edge may be too strict: it indicates an exact sequence of method calls that must be used to construct an object. However, a different object whose value is similar, but which was constructed in a different way, may also be legal and permit the method call to complete non-erroneously. To address this problem, we use a lightweight abstract object profile representation as another argument constraint.

***Abstract Object Profile Constraint.*** For an object, we define its *concrete state* as a vector, $v = \langle v_1, ..., v_n \rangle$, where each $v_i$ is the value of an object field. An abstract object profile maps each field's value to an abstract value. Formally, we use a state abstraction function which maps concrete value $v_i$ to an abstract value $s_i$ as follows:

- Concrete numerical value $v_i$ (of type `int`, `float`, etc.), is mapped to three abstract values $v_i < 0$, $v_i = 0$, and $v_i > 0$.
- Array value $v_i$ is mapped to four abstract values $v_i =$ `null`, $v_i$ is empty, $v_i$ contains null, and all others.
- Object reference value $v_i$ is mapped to two abstract values $v_i$ = `null`, and $v_i \neq$ `null`.
- Boolean and enumeration values are not abstracted. In other words, the concrete value is re-used as the abstract value.

When our tool Palus builds a call sequence model from a sample execution, it computes the abstract object profiles for all arguments and keeps them in the model. Those abstract object profiles prevent the selection of undesirable objects as arguments. During test generation, if Palus is unable to obtain a value created by a given sequence of method calls, then it instead uses one that matches the abstract state profile.

### 3.2 Static Analysis: Model Expansion

The dynamically-inferred model provides a good reference in creating legal sequences. However, the model is only as complete as the observed executions and may fail to cover some methods or method-call invocation orders. To alleviate this limitation, we designed a lightweight static analysis to enrich the model. Our static 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 behaviors and states. Thus, when extending a model sequence, our approach prefers to invoke related methods together.

#### 3.2.1 Method Dependence Analysis

Our approach computes two types of dependence relations: *write-read* and *read-read*.

***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`.

***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. Two methods that have a *write-read* dependence relation may also have a *read-read* dependence relation.

In our approach, we first compute the read/write field set for each method, then use the following strategy to merge the effects of the method calls: if a callee is a private method or constructor, we recursively merge its access field set into its callers. Otherwise, we stop merging. This strategy is inspired by the common coding practice that public methods are a natural boundary when developers are designing and implementing features [16, 19].

#### 3.2.2 Method Relevance Ranking

One method may depend on multiple other methods. We define a measurement called *method dependence relevance* to indicate how tightly each pair of methods is coupled. In the guided test generation phase (Section 3.3), when a method `f` is tested, its most dependent methods are most likely to be invoked after it.

We used the tf-idf (term frequency – inverse document frequency) weighting scheme [13] to measure the importance of fields to methods. In information retrieval, the tf-idf weight of a word $w$ in a document *doc* statistically measures the importance of $w$ to *doc*. The importance increases proportionally to the frequency of $w$'s appearance in *doc*, but decreases proportionally to the frequency of $w$'s appearance in the corpus (all documents). Our approach treats each method as a document and each field read or written as a word.

The *dependence relevance* $W(m_k, m_j)$ between methods $m_k$ and $m_j$ is the sum of the tf-idf weights of all fields, $V_{m_k \to m_j}$, via which $m_k$ depends on $m_j$.

$$W(m_k, m_j) = \sum_{v_i \in V_{m_k \to m_j}} tfidf(v_i, m_k)$$

The intuition is that the dependence relevance between two methods is determined by the fields they both access, as well as these fields' importance to the dependent method.

### 3.3 Guided Random Test Generation

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

Within the given time limit, Palus repeatedly performs one of three following actions: (1) create a new sequence from an enhanced call sequence model root, (2) extend an existing sequence along a randomly-selected model transition, and (3) incorporate the statically-inferred dependence information to append dependent methods to a created sequence. At last, Palus will create additional sequences for methods that are not covered by the expanded model using an existing algorithm described in [23]. All created sequences will be executed reflectively on the fly. Their execution results will be checked against the given testing oracles (in the forms of JUnit theories, Figure 1). Normally-executed sequence will be

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

**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.**

added into the regression test suite, and error-revealing sequences will be output to programmers.

### 3.3.1 Oracle Checking

Palus integrates the JUnit theory framework (which is similar to Parameterized Unit Tests [30]), permitting programmers to write domain-specific oracles. The JUnit theory framework is first described in [26]. 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 intercept 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.

# 4. EVALUATION

## 4.1 Research Questions

We investigated the following two research questions:

**RQ1: Test coverage.** Do tests generated by Palus achieve higher coverage than an existing pure random test generation tool (Randoop), dynamic-random test generation tool (Palulu), and static-random test generation tool (RecGen)? This research question helps to demonstrate how dynamic and static analyses help in improving automated test generation.

**RQ2: Bug finding.** Do tests generated by Palus detect real-world bugs? This research question helps to demonstrate the bug finding ability of our approach.

## 4.2 Subject Programs

We evaluated Palus on six open-source Java programs (Table 1). tinySQL [31] is a lightweight SQL engine implementation that includes a JDBC driver. SAT4J [27] is an efficient SAT solver. JSAP [14] is a simple argument parser, which syntactically validates a program's command line arguments and converts those arguments into objects. Rhino [25] is an implementation of JavaScript, which is typically embedded into Java applications to provide scripting to end users. BCEL [3] is a Java bytecode manipulation library that lets users analyze and modify Java class files. Apache Commons [1] extends the JDK with new interfaces, implementations, and utilities.

## 4.3 Test Coverage

We compare the structural coverage achieved by Palus and three existing tools in Table 2. With the guidance of a legal call sequence model from dynamic analysis and the static dependence information, Palus achieves 35% higher coverage on average.

| Program (version) | Lines of code | Classes | Methods |
|---|---|---|---|
| tinySQL (2.26) | 7672 | 31 | 702 |
| SAT4J (2.2.0) | 9565 | 120 | 1320 |
| JSAP (2.1) | 4890 | 91 | 532 |
| Rhino (1.7) | 43584 | 113 | 2133 |
| BCEL (5.2) | 24465 | 302 | 2647 |
| Apache Commons (3.2) | 55400 | 445 | 5350 |

**Table 1: Subject Programs.**

| Program | Line Coverage% | | | | Inc% |
|---|---|---|---|---|---|
| | Randoop | Palulu | RecGen | Palus | |
| tinySQL (2.26) | 29 | 41 | 29 | **57** | **72%** |
| SAT4J (2.2.0) | 44 | 44 | - | **65** | **47%** |
| JSAP (2.1) | 62 | 70 | 68 | **72** | **8%** |
| Rhino (1.7) | 22 | 25 | 25 | **28** | **14%** |
| BCEL (5.2) | 36 | 39 | 29 | **53** | **53%** |
| Apache (3.2) | 38 | 29 | 28 | **38** | **19%** |
| Average | 38 | 41 | 36 | 52 | **35%** |

**Table 2: Experimental results of comparison between Randoop (a pure random approach), Palulu (a dynamic-random approach), RecGen (a static-random approach) and Palus in terms of line coverage. Column "Inc%" shows the average coverage improvement achieved by Palus. RecGen fails to generate compilable tests for the SAT4J program.**

Specifically, Palus outperforms other tools on 5 out of 6 subjects in test coverage, and achieves the same coverage for the last subject (Apache Commons) with Randoop. There are three primary reasons for this improvement.

First, guided by the inferred model, it is easier for Palus to construct legal sequences. Sequences like creating a database connection in the tinySQL subject, initializing a solver correctly in the SAT4J subject, and generating a syntactically-correct JavaScript program in the Rhino subject all require invoking method calls in a specific order with specific arguments. Such sequences are difficult to create by the randomized algorithms implemented in Randoop or RecGen.

Second, the static analysis used in Palus helps to diversify sequences on a specific legal path by testing related methods together. This helps Palus to reach more program states. For the last subject program, Apache Commons, Palus actually *falls back* to Randoop. The reason is that, Apache Commons is designed as a general library and has very few constraints that programmers must satisfy. For example, one can put any object into a container without any specific method invocation order or argument requirement. Therefore, the information provided by the sample execution trace is not very useful for Palus. Randomly invoking methods and finding type-compatible argument values across the whole candidate domain could achieve the same results.

Third, when comparing with Palulu, we found that, for most subjects, Palulu creates a legal sequence with the assistance of the inferred model. However, the pure random generation phase in Palulu creates many illegal object instances, which pollute the sequence pool. In addition, the model inferred by Palulu lacks necessary constraints for method arguments, so that it is likely to pick up invalid objects from the potentially polluted sequence pool and then fail to reach desirable states. Furthermore, Palulu does not use static analysis to diversify a legal sequence by appending related methods, and may miss some target states

## 4.4 Bug Finding Ability

We demonstrate the bug finding ability of Palus on both six open-

| Subject | Classes | Bugs Found by Palus |
|---|---|---|
| Google Product A | 238 | 4 |
| Google Product B | 600 | 3 |
| Google Product C | 1269 | 2 |
| Google Product D | 1455 | 13 |
| Total | 3562 | 22 |

**Table 3: Four Google products used to evaluate Palus. Column "Classes" is the number of classes we tested, which is a subset of each product. The last column shows the number of distinct bugs found by Palus.**

| Subject Programs | Number of bugs found | | | |
|---|---|---|---|---|
| | Randoop | Palulu | RecGen | Palus |
| tinySQL | 1 | 1 | 1 | 1 |
| SAT4J | 1 | 1 | – | 1 |
| JSAP | 0 | 0 | 0 | 0 |
| Rhino | 1 | 1 | 1 | 1 |
| BCEL | 74 | 70 | 37 | 74 |
| Apache Commons | 3 | 3 | 3 | 3 (4*) |
| *Total* | 80 | 76 | 42 | 80 (81*) |

**Table 4: Bugs found by Randoop, Palulu, RecGen, and Palus in the programs of Table 1. The starred values include an additional bug Palus found with an additional testing oracle written as a JUnit theory. RecGen fails to generate compilable tests for the SAT4J subject.**

source subjects (Table 1) and four well-tested Google products (Table 3).

### *4.4.1 Bugs in Experimental Subjects*

The first part of this evaluation compares the number of unique bugs found by Randoop, Palulu, RecGen, and Palus using default contracts as testing oracles. The default contracts supported by all four tools are listed as follows.

- For any object `o`, `o.equals(o)` returns true
- For any object `o`, `o.equals(null)` returns false
- For any objects $o_1$ and $o_2$, if $o_1$`.equals(`$o_2$`)`, then $o_1$`.hashCode()` == $o_2$`.hashCode()`
- For any object `o`, `o.hashCode()` and `o.toString()` throw no exception

Randoop and Palus found the same number of bugs in all the open-source subjects, while Palulu and RecGen found less. Randoop did just as well as Palus because most of the bugs found in the subject programs are quite superficial. Exposing them does not need a specific method-call sequence or a particular value. For example, a large number of BCEL classes incorrectly implement the `toString()` method. For those classes, a runtime exception will be thrown when calling `toString()` on objects created by the default constructors. On the other hand, tests generated by Palulu are restricted by the inferred model, and thus miss some un-covered buggy methods.

The second part of the experiment evaluates Palus' bug finding ability using additional testing oracles written as JUnit theories. We only 5 simple theories for the Apache Commons Collections library based on our understanding. For example, according to the JDK specification, `Iterator.hasNext()` and all its overriding methods should never throw an exception. Thus, we wrote a theory (Figure 2) for all classes that override the `Iterator.hasNext()` method. After that, we re-ran Palus to generate new tests, and Palus found one new bug. That is, the `hasNext()` method in `FilterListIterator` throws an exception in certain circumstances. This bug has been reported to, and confirmed by, the Apache Commons Collections developers.

### *4.4.2 Palus at Google*

A slightly customized version of Palus (for Google's testing infrastructure) was internally used at Google. The source code of every Google product is required to be peer-reviewed, and goes through a rigorous testing process before being checked into the code base. We chose four products to evaluate Palus (Table 3). Each product has a comprehensive unit test suite. We executed the associated test suite to obtain a sample execution trace, then used Palus to generate tests. The results are shown Table 3.

Palus revealed 22 previously-unknown bugs in the four products. Each bug has been submitted with the generated test. In this case study, we only checked generated tests against default properties as listed in Section 4.4.1, since writing domain-specific testing oracles requires understanding of a specific product's code, which none of the authors has.

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, it is difficult for testers to partition the input space in a way that ensure all important cases will be covered. Testers often miss corner cases. While Palus makes no guarantees about covering all relevant partitions, its combined static and dynamic test generation strategy permit it *learn* a call sequence model from existing tests, *fuzz* on a specific legal method-call sequences, and then lead it to create tests for which no manual tests were written. As a result, Palus discovers many missed corner cases like testing for an empty collection, checking type compatibility before casting, and dereferencing a possibly null reference.

## 5. CONTRIBUTIONS

This paper proposed a combined static and dynamic automated test generation approach, and implemented it in the Palus tool. Our approach is novel in using information from a dynamic analysis to create legal sequences and using information from a static analysis to diversify the generated sequences. Thus, our approach could be regarded as *fuzzing on a specific legal path*. Integration with the JUnit theory framework permits programmers to write projectspecific testing oracles.

A comparison between four different test generation tools on six open-source programs and four Google products demonstrated the effectiveness of our approach. Our experience of applying Palus on Googles code base suggests Palus can be effective in finding real-world bugs.

The source code of Palus and the experimental data on open-source programs are publicly available at:
`http://code.google.com/p/tpalus/`

***Acknowledgements*** We thank anonymous ISSTA'11 reviewers for their insightful feedback on a previous conference version [34].

## 6. REFERENCES

[1] Apache Commons Collections. `http://commons.apache.org/collections/`.

[2] 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.

[3] BCEL project page. `http://jakarta.apache.org/bcel/`.

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

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

[6] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *In Proceedings of Net.ObjectDays 2005*, pages 545–557, 2005.

[7] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.

[8] 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.

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

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. pages 213–223, 2005.

[12] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: Object capture-based automated testing. In *ISSTA 2010*, pages 159–170, July 2010.

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

[14] JSAP home. `http://martiansoftware.com/jsap/`.

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

[16] F. Long, X. Wang, and Y. Cai. API hyperlinking via structural overlap. In *In Proc. ESEC/FSE '09*, pages 203–212, 2009.

[17] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08*, pages 501–510, New York, NY, USA, 2008. ACM.

[18] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of java programs. In *ASE '01*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[19] S. McConnell. Code complete, 2nd edition. Microsoft Press, 2004.

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

[21] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. ECOOP 2005*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

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

[23] 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.

[24] 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.

[25] Rhino project page. `http://www.mozilla.org/rhino/`.

[26] 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 Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, January 14, 2008.

[27] SAT4J project page. `http://www.sat4j.org/`.

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

[29] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE 2009*, August 2009.

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

[31] tinySQL home. `http://www.jepstone.net/tinySQL/`.

[32] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02*, pages 218–228, New York, NY, USA, 2002. ACM.

[33] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA '06*, pages 145–156, New York, NY, USA, 2006. ACM.

[34] S. Zhang, D. Saff, Y. Bu, , and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. 11th International Symposium on Software Testing and Analysis (ISSTA 2011)*, 2011.

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