Combined Static and Dynamic Automated Test Generation

Sai Zhang[1]   David Saff[2]   Yingyi Bu[3]   Michael D. Ernst[1]
[1]University of Washington   [2]Google, Inc   [3]University of California, Irvine
{szhang, mernst}@cs.washington.edu,   saff@google.com,   yingyib@ics.uci.edu

ABSTRACT

In an object-oriented program, a unit test often consists of a sequence of method calls that create and mutate objects, then use them as arguments to a method under test. 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 automated 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 testing approach. 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) on several popular open-source Java programs. Tests generated by Palus achieved higher structural coverage and found more bugs.

Palus is also internally used in Google. It has found 22 previously-unknown bugs in four well-tested Google 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 unit test consists of a sequence of method calls that explores a particular aspect of the behavior of the methods under test. Two primary objectives of testing are to achieve high structural coverage, which increases confidence in the code under test, and to find unknown bugs. To achieve either objective, unit testing requires desirable method-call

sequences (in short, *sequences*) that create and mutate objects. These sequences generate target object states of the receiver or arguments of the method under test.

To reduce the burden of manually writing unit tests, many automated test generation techniques have been studied [2,6,7,19,26,33]. Of existing test generation techniques, bounded-exhaustive [5,23], symbolic execution-based [15, 32, 37], and random [6, 26] approaches represent the state of the art. *Bounded-exhaustive* approaches generate sequences exhaustively up to a small bound on sequence length. However, generating good tests often requires longer sequences beyond the small bound. *Symbolic execution*-based analysis tools such as JPF [37] and CUTE/jCUTE [32] explore paths in the program under test symbolically and collect symbolic constraints at all branching points in 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 the quality of generated inputs heavily depends on the test driver (which is often manually written [15, 32, 37]). *Random* approaches [6, 26]) are easy to use, scalable, and able to find previously-unknown bugs [26], 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 to occur in a certain order with specific arguments. Thus, most randomly-created sequences may be illegal (violate the implicit specification) or may fail to reach new target states.

One possible way to improve the effectiveness of automated testing techniques is to use a formal specification to guide test generation [7, 16, 36]. However, such a specification is often absent in practice. Another way, which we follow, is to combine static and dynamic analysis. 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. Their mutual strengths can enhance one another [13].

This paper presents an automated technique (and its tool implementation called Palus) to generate *legal* and *behaviorally-diverse* tests. Palus operates in three steps: dynamic inference, static analysis, and guided random test generation. (1) In its dynamic component, Palus takes a correct execution as input, infers a call sequence model, and enhances it by annotating argument constraints. This enhanced call sequence model captures legal method-call orders and argument values observed from the sample execution. (2) In its static component, Palus computes dependence relations among the methods under test. The static analysis devised in this paper first identifies field access (read or write) information for each method, then uses the tf-idf weighting scheme [18] to weight the dependence between each pair of methods. In general, methods that read and write the same field are dependent, so testing them together has a

higher chance of exploring different program behaviors. (3) Finally, both the dynamically-inferred model and the statically-identified dependence information guide a random test generator [26] to create legal and behaviorally-diverse tests.

Several past research tools follow an approach similar to ours, but omit one or two of the three stages of our approach. Randoop [26] 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 constraints for method arguments and has no static analysis phase to enrich the dynamically-inferred model with information about methods that are not covered by the sample execution. Finally, RecGen [40] uses a static-random approach. RecGen does not have a dynamic phase, and uses a static analysis to guide random test generation. Lacking guidance from an accurate dynamic analysis, RecGen may fail to create legal sequences for programs with constrained interfaces, and may miss many target states (an example will be discussed in Section 2).

Unlike past research tools [2, 26, 40] mentioned above that only check generalized programming rules (e.g., the symmetry property of equality: `o.equals(o)`), Palus integrates with the JUnit theory framework [30]. This permits programmers to write project-specific testing oracles for more effective bug finding.

Compared to past approaches, Palus increases the structural coverage of generated tests or improves their ability to detect bugs. We evaluated it on six popular open-source applications. The tests generated by Palus achieved much higher structural coverage, and detected more unknown bugs than the other approaches.

Palus is also internally used at Google. It found 22 previously-unknown bugs in four well-tested products. These results suggest that Palus can be effective in detecting real bugs in large codebases.

The source code of Palus is publicly available at: http://code.google.com/p/tpalus/

2. MOTIVATING EXAMPLE

We next give an overview of our approach with an illustrative example taken from the tinySQL [35] project, shown in Figure 1.

A tinySQL test must satisfy both *value* and *ordering* constraints. Creating a `tinySQLConnection` object requires a valid `url` string like `jdbc:tinySQL:fileNameOnDisk` to pass the check at line 4 or line 16[1]; and the `sql` string at line 23 must be a valid SQL statement like `select * from table_name`. Moreover, a legal sequence requires calling methods in a correct order: a valid `tinySQLDriver`, a `tinySQLConnection`, and a `tinySQLStatement` object must be created before a query can be issued. If a test generator fails to create arguments that satisfy these constraints, no `tinySQLConnection` and `tinySQLStatement` object will be created, no database query will be issued successfully, and the generated tests will fail to cover most of the code. As revealed by our experiment in Section 5, a pure random approach Randoop and a static-random approach RecGen could achieved only 29% line coverage for the tinySQL program.

A dynamic-random approach like Palulu [2] takes a (correct) sample execution as shown in Figure 2 as input, builds a call sequence model, and uses the model to guide a random test generator. The model inferred by Palulu from Figure 2 is shown in Figure 3. This model captures possible legal method-call orders (e.g., calling `executeQuery(String)` before `close()`) and argument values (e.g., the observed values of `url` and `sql` from the sample execution) to construct a sequence. After obtaining the model, Palulu uses a two-

[1] Another constructor `tinySQLConnection()` creates an empty object with all fields uninitialized. To become a valid object, its fields need to be assigned separately.

```
1. public class tinySQLDriver implements java.sql.Driver {
2.    public tinySQLDriver() {}
3.    public Connection connect(String url, Properties info) {
4.      if(!acceptsURL(url) {return null;}
5.      return getConnection(info.getProperty("user"),url,this);
6.    }
      ...8 more methods omitted here...
   }
7. public class tinySQLConnection {
8.    protected String url, user, tsql;
9.    protected Driver driver;
10.   public tinySQLConnection() {}
11.   public tinySQLConnection(String user, String u, Driver d)
12.      throws SQLException {
13.      this.url = u;
14.      this.user = user;
15.      this.driver = d;
        //check the validity of url
16.      this.tsql = get_tinySQL();
      }
17.   public Statement createStatement() {
18.      return (Statement)new tinySQLStatement(this);
      }
      ... 20 more methods omitted here...
   }
19. public class tinySQLStatement {
20.   public tinySQLStatement(tinySQLConnection conn) {
21.      this.connection = con;
22.   }
23.   public ResultSet executeQuery(String sql) {
24.    //check the validity of this.connection
25.    //then issue the query and get the result
      }
      ... 20 more methods omitted here...
   }
```

Figure 1: Three classes taken from the tinySQL project [35]. Creating good sequences for this code requires invoking method calls in a specific order with specific arguments.

```
1. tinySQLDriver driver = new tinySQLDriver();
2. Connection conn
     = driver.connect("jdbc:tinysql:db", new Property());
3. tinySQLStatement stmt = new tinySQLStatement(conn);
4. stmt.executeQuery("select * from table_name");
5. stmt.close();
6. conn.close();
```

Figure 2: A sample method call sequence for the code in Figure 1. A dynamic-random approach can use this sample execution to build a legal call sequence model, as shown in Figure 3.

phase strategy to generate sequences. In the first phase, it creates sequences randomly *without* using the model, and keeps all generated sequences in a pool. In the second phase, it creates sequences either by generating a new sequence from a model root (e.g., node A, B, or C in Figure 3) or extending an existing sequence along a model edge. If creating a sequence needs a type T argument, it will *randomly* pick a sequence that creates a type T value from the sequence pool.

Although the Palulu approach offers a possible solution to create legal sequences, it faces several challenges to be effective in practice. First, the random phase in Palulu may end up with few legal sequences. For example, it is very likely to generate no `tinySQLStatement` and `tinySQLConnection` objects for the motivating example. Second, the call sequence model in Figure 3 does not consider constraints for non-primitive arguments. For example, the constructor of `tinySQLStatement` needs to have a `tinySQLConnection` object with all its fields correctly initialized as an argument. However, the Palulu-inferred model does not keep such constraints. Therefore, Palulu will pick an existing `tinySQLConnection` object randomly from the pool, and thus may fail to create a valid `tinySQLStatement` object. Third, the sequences
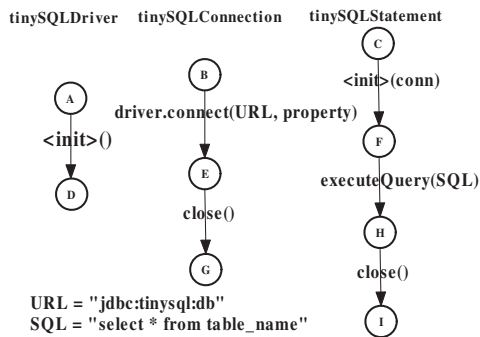
Figure 3: The call sequence model built by a dynamic-random approach (Palulu [2]) for classes in Figure 1 using the sample execution in Figure 2. Nodes A, B, and C represent the root node for each class. Potential nested calls are omitted here.

it generates are restricted by the dynamically-inferred model, which tends to be incomplete. For instance, a correct sample execution would never contain an exception-throwing trace like:

```
tinySQLStatement.close();
tinySQLStatement.executeQuery("select * from ..");
```

Thus, Palulu may fail to generate sequences that either have different method-call orders from the inferred model or contain methods not covered by the sample execution. In our experiment as described in Section 5, although Palulu outperforms Randoop and RecGen, it achieved only 41% code coverage for the tinySQL program.

Our Palus approach remedies the above problems by enhancing the call sequence model with argument constraints and performing a static analysis to find dependent methods that should be tested together (including those not covered by the model).

Take the code snippet in Figure 1 as an example. Palus first constructs a call sequence model (Figure 6) and annotates it with two kinds of constraints, namely, *direct state transition dependence constraints* and *abstract object profile constraints* (Section 3.1.2). These two constraints permit Palus to find a desirable object for creating a legal sequence as well as a group of distinct objects. Then, Palus performs a static analysis to identify method dependence relations based on the fields they may read or write, and prefers related methods when extending a model sequence. Testing methods that access the same field together may have a higher chance to explore different program states. For example, let us assume method `executeQuery` in class `tinySQLStatement` reads field `connection`, and method `close` writes field `connection` (assigns it to `null`). Thus, when testing method `executeQuery`, Palus may recommend to invoke method `close` before it, since testing them together permits reaching a new program state, and checks the behavior of executing a query after closing a connection. This combined static and dynamic test generation strategy permits Palus to generate more effective tests. For the tinySQL program, Palus achieves 59% code coverage, which is 30% higher than Randoop and RecGen and 18% higher than Palulu.

Palus also integrates with the JUnit theory framework [30], which permits programmers to clarify design intentions and write project-specific testing oracles such as the one in Figure 4. Palus will automatically execute this theory and check its output when finding a `tinySQLStatement` object and a `String` object.

3. APPROACH

Palus consists of four components (Figure 5), namely, a load-time instrumentation component and a dynamic model inference component (dynamic analysis, Section 3.1), a static method analysis

```
@Theory
void noActionAfterClose(tinySQLStatement stmt,String sql){
    Assume.assumeNotNull(stmt);
    Assume.assumeNotNull(sql);
    stmt.close();
    try {
        stmt.executeQuery(sql);
        fail("The statement should have been closed!");
    } catch (Exception e) {
        //ok here
    }
}
```

Figure 4: A testing oracle expressed as a JUnit theory. For any non-null `tinySQLStatement` object, the `close` method should not throw any exception, after which `executeQuery` must throw an exception.
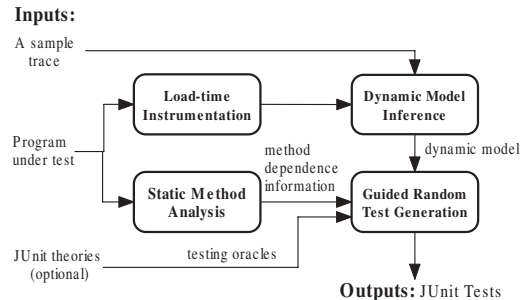

Figure 5: Architecture of the Palus tool.

component (static analysis, section 3.2), and a guided random test generation component (Section 3.3).

The load-time instrumentation instruments the Java bytecode. The dynamic model inference component collects traces from the sample execution and builds an enhanced call sequence model for each class under test. The static method analysis component computes the set of fields that may be read or written by each method, and then calculates pairwise method dependence relevance values. Finally, the dynamically-inferred model and method dependence information together guide the random sequence generation process.

3.1 Dynamic Analysis: Model Inference from Sample Executions

We first briefly introduce the call sequence model in Section 3.1.1, then present our enhancement in Section 3.1.2.

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[2], 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

---

[2]The original model only keeps primitive and string arguments, and we will extend it to other argument types in the next section.
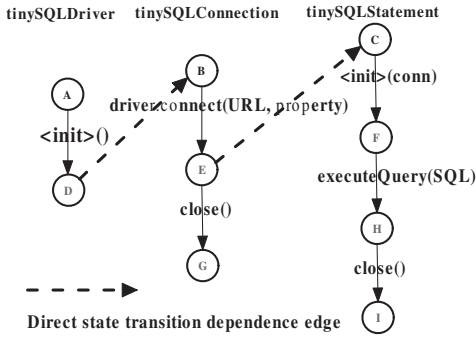
Figure 6: An enhanced call sequence model with direct state transition dependence edges for the example in Figure 1.

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.

Consider the sample execution in Figure 2. The `Driver` object used at line 2 is the result produced by line 1, and the `Connection` object used at line 3 is the execution outcome of line 2. Therefore, our approach adds two *direct state transition dependence* edges to the original call sequence model, and results in an enhanced model shown in Figure 6. Using the enhanced model, when a test generation tool is creating a sequence (for example, calling method `tinySQLStatement(conn)`), it will follow the dependence edge backward to an argument.

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.

For example, let us assume a `tinySQLConnection` object has four fields `url`, `user`, `tsql`, `driver` as in Figure 1. A valid `tinySQLConnection` object might thus have a state v = $\langle$'jdbc:tinysql:file', 'user', 'select * from table', new `tinySQLDriver()`$\rangle$, which corresponds to an abstract state s = $\langle \neq$null$, \neq$null$, \neq$null$, \neq$null$\rangle$.

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. 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.

Even coarse profiles prevent the selection of undesirable objects as arguments. For example, if we run Palulu on the tinySQL subject programs for 20 seconds, it creates 65 `tinySQLConnection` objects. However, only 5 objects have legal state, that is, a state corresponding to the observed abstract state $\langle \neq$null$, \neq$null$, \neq$null$, \neq$null$\rangle$. The remaining 60 objects have the abstract state $\langle =$null$, =$null$, =$null$, =$null$\rangle$. A tool like Palulu that randomly picks a `tinySQLConnection` object is very likely to end with an illegal object instance. In contrast, Palus has a higher chance of getting a legal `tinySQLConnection` object, because it selects a `tinySQLConnection` object with an abstract state $\langle \neq$null$, \neq$null$, \neq$null$, \neq$null$\rangle$.

3.2 Static Analysis: Model Expansion with Dependence Analysis

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 [21, 24].

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 method sequence generation phase (Section 3.3.1), 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 [18] 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*.

Auxiliary Methods:

CoveredMethods(models): return all model-covered methods
DivideTime(timelimit): divide timelimit into 2 fractions
RandomMember(set): return a set member randomly
RandomGenerate(method): create a sequence for method using pure random test generation [26]

GenSequences(timelimit, methods, models, dependences)

```
 1: uncovered_methods ← methods \ CoveredMethods(models)
    {reachingSeqs maps each node to the sequences that reach it}
 2: reachingSeqs ← new Map⟨ModelNode, List⟨Sequence⟩⟩
    {In our experiments, t_model-guided : t_unguided = 4 : 6}
 3: t_model-guided, t_unguided ← DivideTime(timelimit)
 4: seqs ← {}
 5: while t_model-guided not expired do
 6:     Randomly perform one of the following two actions:
 7:     1. root ← RandomMember(models)
 8:        tran ← RandomMember(root.transitions)
 9:        newSeq ← GuidedSeqGen(tran, dependences)
10:        seqs ← seqs ∪ newSeq
11:        reachingSeqs[trans.dest_node].add(newSeq)
12:     2. node ← RandomMember(reachingSeqs.keySet())
13:        seq ← RandomMember(reachingSeqs[node])
14:        tran ← RandomMember(node.transitions)
15:        newSeq ← GuidedSeqGen(tran, dependences)
16:        seqs ← seqs ∪ newSeq
17:        reachingSeqs[node].remove(seq);
18:        reachingSeqs[trans.dest_node].add(newSeq);
19: end while
20: while t_unguided not expired do
21:     method ← RandomMember(uncovered_methods)
22:     seqs ← seqs ∪ RandomGenerate(method)
23: end while
24: return seqs
```

Figure 7: Sequence generation algorithm in Palus. The procedure GuidedSeqGen is shown in Figure 8.

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.

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

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 Test Generation

3.3.1 Sequence Generation

The sequence generation algorithm listed in Figure 7 takes four arguments, namely *a time limit*, *a set of methods* for which to generate sequences, *enhanced call sequence models* constructed from observed sample executions (Section 3.1), and *method dependence relations* (Section 3.2).

The algorithm shown in Figure 7 works in two phases: the first phase for methods that were executed in the sample execution and so appear in the model, and the second phase for other methods. The generator creates sequences incrementally, maintaining a set

Auxiliary Methods:

GetDependentMethod(m): return a method, randomly chosen with probability proportional to its dependence relevance with m
CreateSequenceForConst(value): create a new sequence that produces the given (primitive or string) value
GetSequenceByState(dep_state): get an existing sequence that produces a result of the given dep_state
GetSequenceByProfile(profile): get an existing sequence that produces a result of the given abstract object profile
GetSequenceByType(T): get an existing sequence that produces a result of type T; otherwise, return GetSequenceForConst(null)
Concatenate(param_seqs, m_1, m_2): concatenate param_seqs, m_1, m_2 to form a new sequence

GuidedSeqGen(transition, dependences)

```
 1: m(T_1, ... T_n) ← transition.method
 2: param_seqs ← new List⟨Sequence⟩
 3: for each T_i in (T_1, ..., T_n) do
 4:     if T_i is primitive or string type then
 5:         seq ← CreateSequenceForConst(transition.recordedValue)
 6:     else
 7:         ⟨dep_state, profile⟩ ← transition.constraints
 8:         seq ← GetSequenceByState(dep_state)
 9:         if seq = null then
10:             seq ← GetSequenceByProfile(profile)
11:         end if
12:         if seq = null then
13:             seq ← GetSequenceByType(T_i)
14:         end if
15:     end if
16:     param_seqs ← param_seqs ∪ seq
17: end for
18: dep_m(T'_1, ..., T'_n) ← GetDependentMethod(m)
19: for each T'_i in (T'_1, ..., T'_n) do
20:     param_seqs ← param_seqs ∪ GetSequenceByType(T'_i)
21: end for
22: return Concatenate(param_seqs, m, dep_m);
```

Figure 8: Guided sequence generation algorithm.

of generated sequences (lines 4, 10, 16, and 22). The algorithm also maintains a node-sequence mapping that maps each node in a call sequence model to a list of sequences that end at that node (lines 2, 11, 17, and 18). When generating a sequence, the algorithm either selects an edge that is going out of a model root and creates a sequence for it (lines 7–11), or extends an existing sequence (lines 12–18). After extending an existing sequence (line 15), the existing sequence is removed from the node-sequence map (line 17) and the newly-created sequence is added to the map (line 18). The remove prevents Palus from repeatedly extending a single sequence. The extended sequence actually has had two calls, not one, added to it. It is added to the map as if the second call did not affect its state in the call sequence model; if the assumption is not true, then this addition can add further variability to the generated tests. Finally, to avoid missing model-uncovered methods, our algorithm randomly creates sequences for model-uncovered methods (lines 20–23).

The algorithm in Figure 8 shows steps in creating a sequence using both dynamic model and static dependence information. Given a tested method m (line 1), the algorithm uses constraints (recorded in the dynamic model) to find sequences for its parameters (lines 5, 8, 10, and 13). Then, the algorithm queries the dependence information to get a dependent method dep_m and find parameter sequences for it (lines 19–21). Finally, the parameter sequences,

tested method, and its dependent method are concatenated to form a new sequence (lines 22).

Every sequence returned by GenSequences is executed and checked against a set of generalized properties [26] as well as user-provided oracles (Section 3.3.2). Any violating tests are classified as failure-revealing tests.

3.3.2 Oracle Checking

Our approach integrates the JUnit theory framework [30] (which is similar to Parameterized Unit Tests [34]), permitting programmers to write domain-specific oracles. A theory is a generalized assertion that should be true for any data.

Take the theory in Figure 4 as an example, Palus will automatically check for every non-null tinySQLStatement object and non-null String value pair, 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 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 programmer.

3.3.3 Annotation Support for Additional Inputs

Palus captures the primitive and string type argument values from the observed sample execution in the same way as Palulu does. However, a test generator may still need more additional inputs to explore certain behaviors of the program under test. For example, the sample execution in Figure 2 only covers a special case of *selecting all tuples in the example table* in testing method executeQuery(String). It would be useful if a test generator could feed more different input values (a valid string) to the tested method. Palus provides an annotation called @ParamValue to permit programmers to specify special values for a specific tested method, and will use such user-provided values to create new sequences. For example, a programmer could write an annotation like:

```
@ParamValue(cn="tinySQLStatement",mn="executeQuery")
public static String queryMore = "select * from
    table_name where id > 10";
```

When Palus creates a sequence for the executeQuery method in class tinySQLStatement, it will also use the value of queryMore.

4. IMPLEMENTATION

We implemented Palus using the ASM bytecode analysis framework and the Randoop test generation tool [26]. Palus uses ASM to perform load-time instrumentation of Java bytecode, to record the execution trace. The method dependence analysis is also performed at the bytecode level. Palus works in a fully automatic and push-button way to generate unit tests, and scales to realistic programs. The test generation phase requires from the user a time limit, a sample execution trace, and a list of Java class names. Optionally, the user may specify additional theories for oracle checking (Section 3.3.2), and @ParamValue annotations (Section 3.3.3).

To increase robustness, Palus spawns a test generation process and monitors its execution. If it hangs before the user-specified time, Palus spawns a new process, using a new random seed. Method sequences generated before Palus crashes are also output to users.

5. EVALUATION

We compared tests generated by Palus with tests generated by Randoop (a pure random approach), Palulu (a dynamic-random approach), and RecGen (a static-random approach). We compare the tests' structural coverage and bug detection ability. Finally, we

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.

discuss the suitability of these four tools to programs with different characteristics.

We also report on experience using Palus at Google (Section 5.5.2). Palus found 22 unique previously-known bugs in four well-tested projects from Google's codebase. This experience demonstrates the applicability of Palus to real-world, industrial-strength software products.

5.1 Research Questions

We investigated the following two research questions:

RQ1: Test coverage. Do tests generated by Palus achieve higher structural 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 more bugs than tests generated by Randoop, Palulu, and RecGen? This research question helps to demonstrate the bug finding ability of our approach.

5.2 Subject Programs and Tools

We evaluated Palus on six open-source Java programs (Table 1). tinySQL [35] is a lightweight SQL engine implementation that includes a JDBC driver. SAT4J [31] is an efficient SAT solver. JSAP [20] is a simple argument parser, which syntactically validates a program's command line arguments and converts those arguments into objects. Rhino [28] 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.

The top five subject programs in Table 1 contain many constraints for writing legal tests, while the last subject program (Apache Commons) is designed as a general utility library for developers, and thus has few constraints on its methods. We compare the effectiveness of Palus with three other closely related tools on the above set of subject programs, and investigate the suitability of these four tools on programs with different characteristics.

Randoop implements a technique called feedback-directed random testing [26]. Its key idea is to execute each test as soon as it is generated, and use information obtained from execution to guide the test generation process as it creates further tests. Illegal sequences like sqrt(-1) in which the argument is required to be non-negative will be discarded immediately.

Palulu [2] is a typical dynamic-random approach. Since the existing Palulu implementation relies on a deprecated version of Randoop, we re-implemented its algorithm in Palus[3].

---

[3]Our re-implementation of Palulu achieves higher coverage than the previous one: for example, 41% and 44% line coverage for tinySQL and SAT4J, respectively, compared to 32% and 36% for the previous implementation [2].

Program	Steps to obtain a sample trace	Coverage (%)	
		line	branch
tinySQL	Execute 10 simple SQL statements	40	30
SAT4J	Solve a satisfiable formula of 5 clauses	20	12
JSAP	Run its unit test suite	35	30
Rhino	Interpret a stream of JavaScript code	19	10
BCEL	Parse one class	17	8
Apache Commons	Run a subset (484 tests) of its unit test suite	20	16

Table 2: Steps to obtain a sample execution, and its coverage.

RecGen [40] is a representative of static-random approaches. It uses the result of a static analysis to guide random test generation by recommending closely related methods.

5.3 Experimental Procedure

To answer **RQ1**, we ran Palus, Randoop, Palulu, and RecGen on each subject with a time limit of 100 seconds per class (not per subject program). Increasing the time limit does not noticeably increase coverage for any of the tools. We then ran the generated tests and used Cobertura [8] to measure line/branch coverage.

Palus and Palulu require a sample trace to infer a model. We ran each subject's existing unit test suites, or, if no test suite is available, one simple example from their user manuals. Palus and Palulu use the same execution trace for test generation. Table 2 summarizes the steps to obtain a sample trace, and its coverage.

For **RQ2**, we conducted two separate experiments. First, we executed the JUnit tests generated by the four tools and measured the number of unique bugs they find. Second, we wrote several simple specifications using the JUnit theory framework, and ran Palus on each subject again. We manually checked each failed tests to verify those revealed bugs. Finally, we compared the bug-finding ability of the four tools on four well-tested Google products.

5.4 Coverage Results

Figure 9 compares Randoop, Palulu, RecGen, and Palus on the subject programs.

5.4.1 Comparison with a Pure Random Approach

Palus outperforms Randoop on 5 out of 6 subjects in both line and branch coverage, and achieves the same coverage for the last subject (Apache Commons). There are two 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 algorithm implemented in Randoop. In fact, most of the tests generated by Randoop for the first five subjects in Figure 1 only cover error-handling statements/branches.

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 in-

formation 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.

5.4.2 Comparison with a Dynamic-Random Approach

Tests generated by Palus achieve higher line/branch coverage than Palulu for all six subjects. For most subjects, Palulu creates a legal sequence with the assistance of the inferred model. However, there are three major reasons for the result difference. First, the pure random generation phase in Palulu creates many illegal object instances, which pollute the sequence pool. Second, 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. Third, Palulu does not use static analysis to diversify a legal sequence by appending related methods, and may miss some target states.

For example, in SAT4J, Palus achieves 81% line coverage and 43% branch coverage for the `org.sat4j.minisat.constraints.cnf` package, while Palulu only achieves 39% and 21%, respectively. The reason is that class constructors in this package often need specific argument values. For instance, the constructor of class `BinaryClause` only accepts an `IVetInt` (an integer vector wrapper) object with exactly 2 elements. Thus, without explicit argument constraints, choosing an arbitrary `IVetInt` value could easily fail the object construction. We also find the static analysis phase to be useful. For example, SAT4J's `DimacsSolver.addClause(IVecInt literals)` method first checks a boolean flag `firstConstr`, then takes different branches according to its value. When testing this method, Palus identifies another method `reset()`, which sets the `firstConstr` be `false`, as dependent. Palus invokes `reset()` before calling `addClause(IVecInt literals)` in some tests to make the generated sequences reach more states.

Palus achieves little coverage increase over Palulu in two subjects, namely JSAP and Rhino. We identified two possible reasons. First, JSAP does not have as many constraints as other subjects in creating legal sequences. The only constraints are to check the validity of user inputs from command line before parsing (e.g., users need to provide a well-formatted string input). Moreover, most methods in JSAP perform string manipulation operations, and do not have much dependence between each other. Thus, Palus achieves the same coverage in most of the non-constrained code. Second, over 90% of the source code in Rhino belongs to the JavaScript parser and code optimizer modules. Code in these two modules will not be executed unless the interpreter is provided with a program with certain code structure. Only specific code structures trigger the optimizer to perform certain code transformations. However, the example program we used in the experiment to obtain an execution is a simple JavaScript program without any sophisticated structures. Therefore, the model Palus inferred failed to guide the test generator to cover most of the sophisticated code transformation and optimization part.

5.4.3 Comparison with a Static-Random Approach

Tests generated by Palus achieve higher coverage than RecGen for all subjects. For subjects tinySQL and BCEL, Palus almost doubles the coverage. The primary reason is that, although RecGen uses a well-designed static analysis to recommend related methods to test together, it fails to construct legal sequences for most of the subjects with constrained interfaces.

Another problem is that the latest release of RecGen is built on top of a deprecated version of Randoop that contains bugs that have
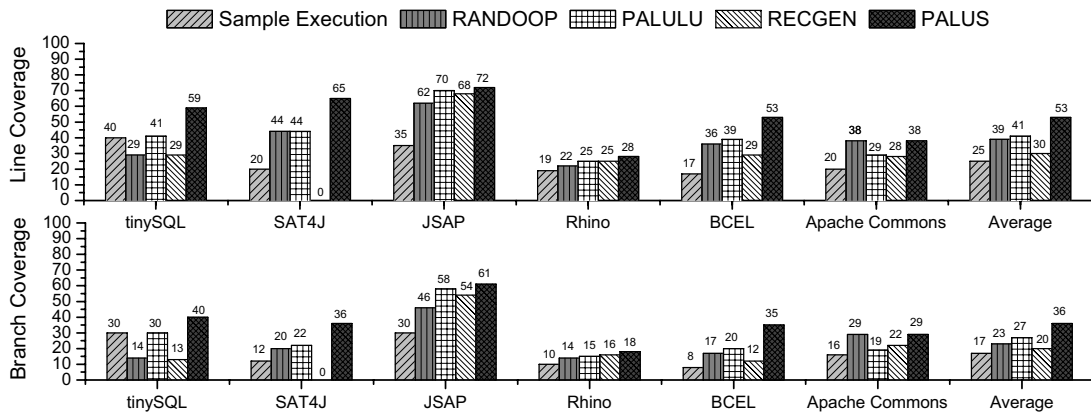
Figure 9: Structural coverage of tests generated by Randoop, Palulu, RecGen, and Palus. For each subject, coverage of the sample execution is also shown. RecGen fails to generate compilable tests for SAT4J. The average coverage results are shown at the far right.
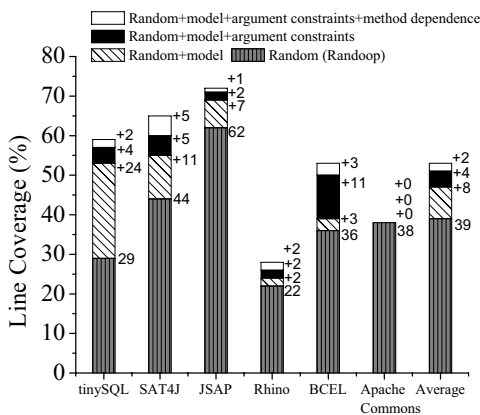

Figure 10: Breakdown of the line coverage increase for 6 subjects in table 1. The average result is shown at the far right. Branch coverage shows a similar pattern, which we omit for brevity.

since been fixed. Thus, RecGen fails to generate compilable tests for the SAT4J subject.

5.4.4 Coverage Increase Breakdown

Figure 10 shows the line coverage increase contributed by each component in Palus. Using a call sequence model in Palus's algorithm (Figure 8) improves the line coverage percentage by 0–24%, compared to a random approach. Enhancing the call sequence model with argument constraints improves line coverage percentage by another 0–11%. The static method dependence analysis further improves line coverage percentage by 0–5%. Note that in Figure 10, Random + model is better than Palulu [2], since the pure random generation phase in Palulu pollutes the sequence pool.

Overall, by combining dynamic inference and static analysis, Palus improves the line coverage on average by 14% (min = 0% for Apache Commons, max = 30% for tinySQL) over a pure random approach.

5.4.5 Input Execution Trace and Test Coverage

We next investigated how sensitive are the test coverage results of Palus to its input execution traces. We chose tinySQL and SAT4J as two examples, since the input trace size can be easily controlled. For tinySQL, we executed all the available SQL statements (23 in total) from its user manual to obtain a more comprehensive trace,

and then used that to guide Palus. The generated tests achieved slightly higher structural coverage (60% line coverage and 41% branch coverage) than the results reported in Figure 9 (59% line coverage and 40% branch coverage). For SAT4J, we obtained two additional execution traces by solving two complex formulas from its example repository [31], containing 188 clauses and 800 clauses, respectively. Using the formula with 188 clauses, Palus achieved 65% line coverage and 37% branch coverage. Using the formula with 800 clauses, Palus achieved 66% line coverage and 37% branch coverage. Compared to the results in Figure 9 by using a formula with 5 clauses (65% line coverage and 36% branch coverage), the coverage increase was slight. Although we have not tried all available examples in SAT4J's code repository, which contains millions of clauses in total, these two experiments indicate that a simple trace is still useful for Palus, and suggest that Palus's results is not very sensitive to its input execution trace.

5.4.6 Model Size and Test Coverage

We further investigated the trade-off between model size and generated test coverage, by applying the *kTail* algorithm [4] to refine Palus's model. The *kTail* algorithm merges model nodes whose equivalence is computed by comparing their tails of length k, where the tails of length k are the method invocation sequences that exit the model node and include at most k method invocations.

We chose $k = 3$ in our experiment to generalize the inferred model, since a value between 2 and 4 is recognized as a reasonable choice in the absence of additional information [22]. We then used the refined model to guide Palus. Figure 11 shows the experimental results. The *kTail* algorithm can substantially reduce the model size by 9–95%, but also decreases the test coverage by 2–18%.

The purpose of a dynamically-inferred model in Palus is to guide a test generator to create legal method-call sequences, not for human inspection to understand program behavior like [11, 22]. Thus, the model size is not crucial to Palus.

5.5 Bug Finding Ability

We compared the bug finding ability of each tool on both six open-source subjects (Table 1) and four well-tested Google products (Table 3). Table 4 gives the experimental results.

5.5.1 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.
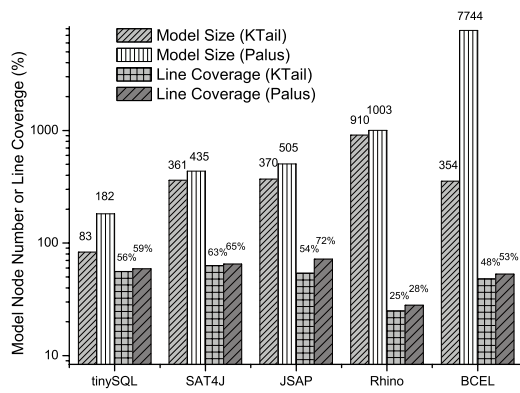
360

Figure 11: Model size (node number in log scale) and test coverage after applying the *kTail* algorithm with k = 3. The line coverage data for Palus is taken from Figure 9.

Subject	Classes
Google Product A	238
Google Product B	600
Google Product C	1269
Google Product D	1455

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.

- 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. Since none of the authors is familiar with the subjects, we only wrote 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 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.

5.5.2 Palus at Google

A slightly customized version of Palus (for Google's testing infrastructure) is internally used at Google. The source code of every Google product is required to be peer-reviewed, and goes through

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*)

Google Products	Number of bugs found			
	Randoop	Palulu	RecGen	Palus
Product A	2	2	–	4
Product B	2	2	–	3
Product C	2	2	–	2
Product D	13	12	–	13
Total	19	18	–	22

Table 4: Bugs found by Randoop, Palulu, RecGen, and Palus in the programs of Tables 1 and 3. 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, and can not be configured to work properly in Google's testing environment.

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 in the bottom half of Table 4.

Palus revealed 22 previously-unknown bugs in the four products, which is more than Randoop and Palulu found within the same time using the same configurations. Each bug has been submitted with the generated test. As of April 2011, 8 of the reported bugs had been fixed. Another 4 bugs were tracked down, and confirmed in the code of an internal code generation tool. The team building the code generation tool has been notified to fix them. Palus found those bugs in 5 hours of CPU time and 5 hours of human effort (spread among several days, including tool configuration and inspection of generated tests). In this case study, we only checked generated tests against default properties as listed in Section 5.5.1, 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. As a comparison, the primary reason why Randoop and Palulu found fewer bugs is that they failed to construct good sequences for some classes under test due to argument constraints.

5.6 Experiment Discussion and Conclusion

Still uncovered branches. Palus achieves coverage far less than 100%. There are several reasons for this. First, the code under test

often includes complex branches that are quite difficult to cover. As mentioned in Section 5.4.2, the code in Rhino's JavaScript parser and optimizer modules is only covered when provided a JavaScript program with certain structure. However, such an input is difficult to generate automatically. Using the `ParamValue` annotation described in Section 3.3.3 could help alleviate this problem. Second, the generated tests depend on the completeness of the inferred model. For parts of the code with constrained interfaces, if a sample trace does not cover that, it is difficult for Palus to create effective tests.

Threats to validity. As with any empirical study, there are threats to the validity of our conclusions. The major threat is the degree to which the subject programs used in our experiment are representative of true practice. Another threat is that we only compared Palus with three state-of-the-art test generation tools. To the best of our knowledge, these are the best publicly-available test generation tools for Java. Future work could compare with other tools, such as JCrasher [6].

Analysis cost. Palus runs in a practical amount of time. Our evaluations were conducted on a 2.67GHz Intel® Core™2 PC with 12GB physical memory (3GB is allocated for JVM), running Ubuntu 10.04 LTS. For the largest subject, Apache Commons, Palus recorded over 300MB of traces from executing its test suite. Palus finished building the enhanced call sequence graph and performing static analysis in 30 seconds. The performance has not been tuned, and we believe that it could be improved further.

Applicability of different techniques. Feedback-directed random test generation (Randoop) is effective in generating tests for programs with few constraints. Palulu improves Randoop's effectiveness on programs with constrained interfaces, but may miss some states for program with dependence relations between methods. RecGen improves Randoop's effectiveness on programs with dependence relations between methods, but could fail to generate valid tests for programs with constrained interfaces. Palus improves on both sides. It may be most effective in generating test inputs for programs with constrained interfaces and inter-method dependences.

Conclusion. The combined static and dynamic test generation technique used in Palus permits generating more effective tests, and finding more bugs in real-world code, compared to previous random test generation techniques.

6. RELATED WORK

We next discuss the most closely-related work in program behavior model inference and automated test generation techniques.

Program Behavior Model Inference. A rich body of work has been done on program behavior model (or specification) inference from either source code [38] or dynamic executions [9, 14, 22]. The concept of learning models from actual program runs was pioneered in [9] by applying a probabilistic NFA learner on C traces. Their approach relies on manual annotations to relate functions to objects. Dynamic invariant detection techniques [14] express properties of data that hold at specific moments during the observed executions. The work by Whaley et al. [38] mines models with anonymous states and slices models by grouping methods that access the same fields. Later, Lorenzoli et al. [22] 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 call sequence 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`).

The abstract object profile we used to enhance the original call sequence model is closely related to the ADABU model [10, 11].

The difference is that the ADABU model [11] classifies methods into *mutators* (methods that change the object state) and *inspectors* (methods that keep the state unchanged), and then uses the return value of each *inspector* to represent an object state. Furthermore, the ADABU model is designed for program behavior understanding. It is highly generalized from the observed executions, and does not represent information on how to construct a legal method-call sequence or how to find desirable argument objects [27], thus can not be directly used for test generation.

Automated Test Generation. Many automated test generation techniques for object-oriented programs [6, 7, 25, 26] have been proposed in the last decade. For example, JCrasher [6] creates test inputs by using a parameter graph to find method calls whose return values can serve as input parameters. Eclat [25] and Randoop [26] use feedback information as guidance to generate random method-call sequences. AutoTest [7] 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 [33] 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. The RecGen [40] approach takes a similar strategy in recommending related methods for sequence creation. It was proposed independently of the static analysis part of Palus [39]. Compared to RecGen, Palus uses a method from the information retrieval community to rank the dependence relevance between pairwise methods, while RecGen mainly considers the degree of overlap of access fields between two methods.

Another two alternative approaches to create test input objects are via direct heap manipulation and using capture and replay techniques. Korat [5] and TestEra [23] 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 inputs. 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 [17] 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 our 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). Test carving [12] and test factoring [29] also use capture and replay techniques, but their focus is orthogonal to ours. They capture the interactions between the target unit and its context to create the scaffolding to replay those interactions. These two techniques focus

on generating small unit tests from system test cases, instead of creating new tests to increase coverage and find new bugs.

The JUnit theory structure [30] can be viewed as a Java implementation of Parameterized Unit Tests (PUT) [34]. Palus uses the theory as an oracle to check the intended program behavior and does not try to convert generated sequences into a PUT.

7. CONCLUSION AND FUTURE WORK

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 project-specific 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 Google's code base suggests Palus can be effective in finding real-world bugs in large, well-tested products.

For future work, we are interested in exploring two research directions. First, we plan to use different models like ADABU [11] to guide automated test generation. Second, we plan to develop machine learning techniques to complement the dynamically-inferred model we introduced here.

8. REFERENCES

[1] Apache Commons Collections project page. 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*, Oct, 2006.

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

[4] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, 21(6), 1972.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, 2002.

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

[7] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *In Proceedings of Net.ObjectDays*, 2005.

[8] Cobertura home. http://cobertura.sourceforge.net/.

[9] 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.

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

[11] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, 2006.

[12] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *SIGSOFT/FSE-14*, pages 253–264, 2006.

[13] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA*, 2003.

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

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[16] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, 2002.

[17] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object capture-based automated testing. In *ISSTA*, 2010.

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

[19] M. Jorde, S. Elbaum, and M. B. Dwyer. Increasing test granularity by aggregating unit tests. In *ASE*, 2008.

[20] JSAP home. http://martiansoftware.com/jsap/.

[21] F. Long, X. Wang, and Y. Cai. API hyperlinking via structural overlap. In *ESEC/FSE*, 2009.

[22] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, 2008.

[23] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[24] S. McConnell. Code complete, 2nd ed. Microsoft Press, 2004.

[25] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, 2005.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[27] Valentin Dallmeier. Personal Communication, May 2011.

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

[29] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE*, 2005.

[30] 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, Cambridge, MA, 01/14, 2008.

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

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

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

[34] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE-13*, 2005.

[35] tinySQL home. http://www.jepstone.net/tinySQL/.

[36] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *ICSM*, 1993.

[37] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.

[38] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002.

[39] S. Zhang, Y. Bu, X. Wang, and M. D. Ernst. Dependence-guided random test generation. *CSE 503 Course Project Report, University of Washington. URL: www.cs.washington.edu/homes/szhang/gencc.pdf*, May 2010.

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