GRT: Program-Analysis-Guided Random Testing

Lei Ma*, Cyrille Artho†, Cheng Zhang§, Hiroyuki Sato*, Johannes Gmeiner‡ and Rudolf Ramler‡
* University of Tokyo, Japan {malei, schuko}@satolab.itc.u-tokyo.ac.jp
†AIST / ITRI, Japan c.artho@aist.go.jp
§ University of Waterloo, Canada c16zhang@uwaterloo.ca
‡Software Competence Center Hagenberg, Austria {johannes.gmeiner, rudolf.ramler}@scch.at

***Abstract*—We propose Guided Random Testing (GRT), which uses static and dynamic analysis to include information on program types, data, and dependencies in various stages of automated test generation. Static analysis extracts knowledge from the system under test. Test coverage is further improved through state fuzzing and continuous coverage analysis. We evaluated GRT on 32 real-world projects and found that GRT outperforms major peer techniques in terms of code coverage (by 13 %) and mutation score (by 9 %). On the four studied benchmarks of Defects4J, which contain 224 real faults, GRT also shows better fault detection capability than peer techniques, finding 147 faults (66 %). Furthermore, in an in-depth evaluation on the latest versions of ten popular real-world projects, GRT successfully detects over 20 unknown defects that were confirmed by developers.**

***Keywords*—Automatic test generation, random testing, static analysis, dynamic analysis**

I. Introduction

A *unit test* for an object-oriented program consists of a sequence of method calls. Manually crafting test sequences is labor-intensive. *Random testing* automatically generates test sequences to execute different paths in a method under test (MUT) [25]. To optimize coverage of test cases, feedback-directed random testing (FRT) [42], [44] uses information generated in earlier iterations of test generation to direct latter iterations. Techniques adopting FRT, such as Eclat [42] and Randoop [44], incrementally build more and longer test sequences by randomly selecting an MUT and reusing previously generated method sequences (that return objects) as input to execute the MUT until a time limit is hit.

While having greatly improved random testing, FRT still suffers low code coverage in many cases [23], [55], [65], [66]. With the advancement of other testing techniques (e. g., search-based testing [16]), random testing and FRT seem to become less competitive [5], [50], [32], [19]. We show that combined static and dynamic analysis can guide random testing and significantly improve its effectiveness.

In this paper, we propose *Guided Random Testing* (GRT). GRT extracts both static and dynamic information from the software under test (SUT) and uses it to guide random testing. GRT works in two phases: (1) A static analysis over the classes under test (CUT) extracts knowledge, such as possible constants during execution, method side effects, and their dependencies. Based on these analysis results, GRT creates comprehensive pools of initial constant values and determines the properties of methods that form the basis of method sequence generation. (2) At run-time, the static information is intelligently combined with dynamic feedback, such as exact type information and test coverage information, to support demand-driven object construction and to guide testing to those MUTs with low coverage.

We implemented the proposed approach of GRT as a test generation tool based on the random testing framework of Randoop [44]. GRT is fully automatic and does not require input specifications or existing test cases. We perform a thorough evaluation of GRT on a large set of benchmarks containing 32 popular real-world applications. The experiments demonstrate the effectiveness of GRT with respect to code coverage, mutation score, and the ability to detect real, known and unknown defects in open source projects. Furthermore, GRT obtained the highest overall score in a contest of automatic test tools, competing with six other well-known testing tools [50], [36]. In summary, this paper makes the following contributions:

1) We propose GRT, a fully automatic testing technique using six collaborating components that extract and use static and run-time information to guide test generation.
2) We evaluate GRT on 32 real-world programs in terms of code coverage and mutation score, comparing it with major peer techniques (i. e., Randoop [44] and EvoSuite [19]) by using multiple time budgets and scenarios.
3) We investigate the defect detection ability of our proposed technique on real bugs in Defects4J [31], [32].
4) We perform an in-depth investigation of the usefulness of GRT in detecting new, previously unknown bugs on ten widely used open source projects. GRT successfully found **23** unknown (and now confirmed) bugs.

This paper is organized as follows: Section II provides relevant background information and presents an overview of GRT. Section III describes each of GRT's components in detail. Section IV shows the evaluation results. Section V compares GRT with related work and Section VI concludes and discusses future work.

II. Background and Overview

A. Random Testing

The general process of software testing consists of three major steps: *creating test inputs*, *executing tests*, and *checking test outputs*. Test automation techniques aim at automating one or more of these steps.

A software under test is often called an *SUT* for short. Similarly, class and method under test are abbreviated to *CUT* and *MUT*, respectively. Testing an MUT with method signature $m(T_{in_1}\, v_1, T_{in_2}\, v_2, \ldots, T_{in_n}\, v_n) : T_{out}$ requires creating objects with types of $T_{in_1}, \ldots, T_{in_n}$ as the inputs of m, and

TABLE I. OVERVIEW OF GRT PROGRAM ANALYSIS COMPONENTS, AND THEIR USAGE FOR TESTING GUIDANCE.

Component	Static / dynamic	Description
Constant mining	static	Extract constants from SUT for both global usage (seed the main object pool) and for local usage as inputs for specific methods.
Impurity	static + dynamic	Perform static purity analysis for all MUTs. At run-time, fuzz selected input from the object pool based on a Gaussian distribution and purity results.
Elephant brain	dynamic	Manage method sequences (to create inputs) in the object pool with the exact types obtained at run-time.
Detective	static + dynamic	Analyze the method input and return type dependency statically, and construct missing input data on demand at run-time.
Orienteering	dynamic	Favor method sequences that require lower cost to execute, which accelerates testing for other components.
Bloodhound	dynamic	Guide method selection and sequence generation by coverage information at run-time.

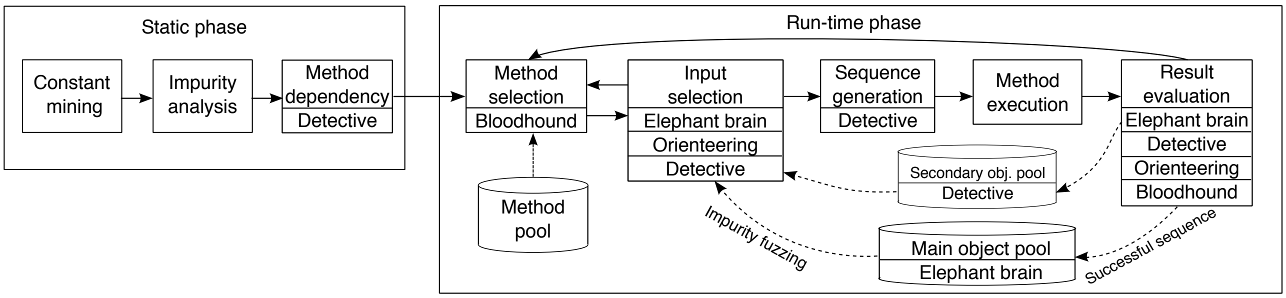

Fig. 1. The workflow overview of guided random testing (GRT), which combines information from static analysis with run-time guidance.

the execution of m returns an object with type T_{out}. The returned object can be further used as input to test another method that requires an argument of type T_{out}. A method sequence in testing consists of a sequence of statements $mSeq = \{s_1, s_2 \ldots, s_n\}$, where each $s_i \in mSeq$ is either an assignment statement $v_{out} = v_{in}$ or method invocation statement $v_{out} = m(v_{in_1}, \ldots, v_{in_n})$ that invokes m with inputs $v_{in_1}, \ldots, v_{in_n}$ and assigns the output to variable v_{out}.

In the context of testing object-oriented programs, test inputs are either primitive values or objects with particular states. To construct useful object states, object-oriented testing often starts with a set of primitive values, and uses them as arguments for specific constructors or methods. In this way, the object states are not directly specified as a set of (primitive) values, but created through the combination of initial values and the execution of method sequences. Objects obtained from a method sequence can be used as input for other methods. When using such an approach, method sequences are conceptually equivalent to input objects.

It is usually impossible to exhaustively enumerate all possible initial values and combinations of method calls. Various methods, such as systematic white-box testing and search-based testing, are proposed to select or create a relatively small number of initial values and method sequences. One of the early ideas is *random testing* [25], which feeds the SUT with randomly generated inputs. It is easy to use, straightforward to automate, and scalable. Random testing has been found effective in detecting program errors [15], [12], [41], [44]. However, randomness without additional guidance is not optimal in practical settings, where testing resources are often limited. A number of techniques have been proposed to improve the effectiveness of random testing using extra information, such as run-time feedback, to control the test generation process, while allowing certain degrees of randomness [44], or to study method sequence patterns from manually written test cases to guide random test generation [65].

B. Guided Random Testing

GRT leverages *knowledge* extracted from the software under test to guide each step of run-time test generation. As shown in Fig. 1, the overall process of GRT begins with extracting constant values from classes under test through a lightweight static analysis. The extracted constant values are used throughout the entire process as "seeds" to create complex object states. Furthermore, a static purity analysis (see Section III-B) categorizes all MUTs into *pure* and *impure* methods. The result of the purity analysis is used to generate unseen object states efficiently. The third static analysis focuses on dependencies between parameter types of methods (input types) and return types of methods (output types). The purpose is to identify the types of objects that are essential for testing MUTs. Since exact types may be determined only at run-time, GRT also performs dynamic analysis to capture type dependencies.

GRT's run-time phase is executed in two or more iterations. In each iteration, run-time information is collected to guide subsequent iterations. The first step of each iteration is selecting a method to be tested from all MUTs in the *method pool*. GRT guides the method selection using code coverage information obtained during the test execution of previous iterations. For the execution of the selected MUT, GRT chooses method inputs from two object pools. Method inputs are maintained in the form of generated method sequences. The main object pool contains method sequences that have been successfully executed in previous iterations, while the secondary object pool contains method sequences generated on demand. When selecting input objects, GRT takes the cost of creating each object (i. e., the cost of executing the corresponding method sequence) into account. Costs are extracted from executions in previous iterations. After the necessary inputs have been selected, GRT combines MUTs with their inputs to generate new method sequences. These method sequences are executed to test the SUT. The execution completes the current iteration of the run-time phase. GRT continues with further iterations

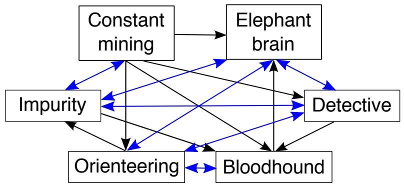

Fig. 2. GRT component interaction and mutual enhancement.

until certain stop criteria (e. g., test coverage) are met or the test time budget is exhausted.

GRT consists of six collaborative components, each of which is briefly described in Table I. We use the component names shown in Table I for brevity. The components closely work together as they extract useful static and dynamic information at specific points and then pass it to other components to facilitate their tasks. The overall effectiveness of GRT results not only from the individual components, but also from their orchestration.

Mutual enhancements between different components are summarized in Fig. 2. An arrow from one component to another signifies that the former enhances the latter; a double arrow (shown in blue) shows mutual benefits. *Constant mining* improves the diversity of the initial object pool by using constants extracted by static analysis. *Impurity* boosts the effect of constant mining through input fuzzing to create objects with more diverse states. *Elephant brain* further diversifies the input object types by using dynamic type information to find objects that cannot be generated using static type information alone. *Detective* constructs necessary objects that cannot be created from the original fixed MUT pool on demand. *Orienteering* accelerates the overall testing speed of GRT and makes the effect of the other components more apparent. Finally, *Bloodhound* intelligently selects MUTs that are not well covered. Upon covering more code of an MUT, more program states are reached, which potentially creates more diverse objects to cover even more code. The next section presents each component in detail.

III. Program Analysis Techniques of GRT

A. Constant Mining: Automatic Constant Extraction

Primitive types (booleans, numbers, characters, and strings) are the basis of creating complex objects. However, values chosen purely at random often fail to satisfy branch conditions. Consider the example from class `PatternOptionBuilder` in Fig. 3: Branches of method `getValueClass` are not covered by Randoop [42], as Randoop does not start with the required predefined primitive values, and is unable to derive the right values that satisfy these branch conditions at run-time. Although manual constant selection is helpful in covering such branches (e. g., as an option in Randoop), it requires much human effort.

To obtain relevant input values without incurring too much overhead, we perform a lightweight static analysis, called *constant mining*. Our key observation is that many useful constant values are used as instruction operands. Constant mining extracts constants from the classes under test and performs constant propagation and constant folding [40] to compute input values as candidates for the initial value pool.

Practical software usually contains a large number of constants, and a constant may only be related to specific branches. Simply selecting the extracted constants as inputs (at random) for all MUTs is of little help to cover specific branches. In addition, putting irrelevant constants in the value pool can increase the overhead for test sequence generation and decrease the overall performance. Therefore, we use the extracted constants on two levels: on a global level (among all classes) and on a local level (a constant is only used for the class containing it).

For global usage, we prioritize the extracted constants by weighting them according to their frequency. The weight is computed based on a modified version of term frequency-inverse document frequency (TF-IDF), which is often used to measure the importance of a term in a set of documents [37]. In the context of constant mining, we treat each class of the SUT as a document and each extracted constant as a term resulting in the weight

$$\text{tf-idf}_v(t, D) = \text{tf}(t, D) * \log \frac{|D| + 1}{|D| + 1 - |d \in D : t \in d|}$$

Here, $\text{tf}(t, D)$ represents the frequency of a constant t occurring in a set of classes D. $|D|$ is the total number of classes in the SUT, and $|d \in D : t \in d|$ is the number of classes that contain the constant t. The formula favors a constant if it is used more frequently and in more classes. Each constant has the chance to be picked, although the selection probability is lower for a constant with smaller weight.

Some constants are used locally as they may be only relevant to methods of a class that contains the constants. In this case, we register each extracted constant for the classes containing it, and select the constants by a predefined probability (noted as p_{const}) as inputs for MUTs of the corresponding class. To obtain even more relevant values, we also use state fuzzing (see Section III-B).

B. Impurity: Purity-Based Object State Fuzzing

In order to generate sequences with a broad variety of object states, we randomly alter (or *fuzz*) the states of existing input objects and pass the fuzzed results to the MUT. We handle primitive numbers based on a Gaussian distribution and non-primitive objects based on method purity analysis.

1) Primitive Value Fuzzing: Primitive inputs are either extracted by constant mining or from method sequence execution results at run-time. To cover a wider range of inputs, we use a heuristic: given values are already close to satisfying some of the branch conditions. When a primitive number c is selected as an input, we adopt a Gaussian distribution to probabilistically fuzz its value and use the altered result as input. Specifically, we use the original value of c and a predefined constant as the mean value μ and standard deviation σ, respectively. We use a Gaussian distribution because it creates new values following our heuristic in that it gives higher probabilities to generate values closer to μ (68.3 % of fuzzed values probabilistically lie in $[\mu - \sigma, \mu + \sigma]$), while still generating values distant from μ.

To fuzz a string value, we randomly choose a string operation among inserting a character, removing a character, replacing a character, and taking a substring of the given string.

2) Purity-Analysis-Based Object State Fuzzing: To test a method m, GRT selects the input objects of m from the previously generated sequences stored in the object pool. To obtain inputs with more diverse object states, we fuzz non-primitive objects by identifying and using methods that have side effects that alter the state of the receiver instance, method arguments, or (global) static fields.

Method purity analysis [54] classifies MUTs into pure and impure methods. Methods without side effects are *pure*, and methods with side effects are *impure*. Only impure methods can change the state of an object [54]. While invoking pure methods is useful to check object states, selecting such methods often creates long and redundant sequences where object states stay unchanged, slowing down overall growth of coverage. Therefore, impure methods are favored over pure methods in order to frequently mutate object states to satisfy more branch conditions.

Given an input object o_i of type T_i, we perform static purity analysis to gather all impure methods that can change the state of an object (reference) of type T_i. Among these impure methods, we randomly select a method $m(T_1\, o_1, \ldots, \boldsymbol{T_i\, o_i}, \ldots, T_n\, o_n)$ at run-time, and invoke m on o_i to fuzz the state of o_i (each of the impure methods can be selected multiple times to fuzz different input objects). Since m may also require other input types, we first search and reuse such objects from the method sequence of o_i, and select the remaining missing objects from the object pool. The fuzzed object of o_i is then passed as input to test the target method. For example, when testing class `List`, impure methods such as `add(element)` and `remove(element)` are used to fuzz a `List` object l for more states. Static [54], [58], [26] and dynamic [59], [64] purity analysis techniques exist. We adopt a static technique [26] to avoid additional overhead at run-time.

C. Elephant Brain: Dynamic Input Sequence Management

Subtyping is pervasive in object-oriented programs. An object reference obj of type T can be assigned to another reference of its super type T' such as $T'\, obj' = obj$, which makes the usage of the object referenced by obj conform to the interface of T'. Such an assignment brings benefits of simplifying the interface by allowing diverse implementations through dynamic binding. However, it poses a challenge to test case generation, since the exact (run-time) type of an object may not be the same as its declared type. This limits many existing automatic testing techniques that adopt a static type-based method sequence management [12], [42], [44], [65].

In the example shown in Fig. 3, branch coverage in method `createVal` requires both suitable primitive values and a class descriptor returned by `getValueClass` method. However, static type management stores the object o returned by `getValueClass` only as the type `Object` according to its declaration. An instance of the type `Object` cannot be used as the input for `createVal` that requires the argument of the type `Class`, unless it is aware that the dynamic type of o is compatible with (or can be used as) the type `Class` and

```
1  package org.apache.commons.cli;
2  public class PatternOptionBuilder{
3    public static final Class STRING_VAL=String.class;
4    public static final Class OBJECT_VAL=Object.class;
5    public static final Class NUMBER_VALUE = Number.class;
6    // 6 more similar fields omitted.
7    public static Object getValueClass(char ch) {
8      switch (ch) {
9       case '@':return PatternOptionBuilder.OBJECT_VAL;
10      case ':':return PatternOptionBuilder.STRING_VAL;
11      case '%':return PatternOptionBuilder.NUMBER_VALUE;
12      // 6 more case branches omitted.
13     }
14     return null;
15 } } // 2 more methods omitted.
16 public class TypeHandler {
17   // 1 method omitted.
18   public static Object createVal(String s, Class c) {
19     if (PatternOptionBuilder.STRING_VAL == c)
20       return s;
21     else if (PatternOptionBuilder.OBJECT_VAL == c)
22       return createObject(s);
23     else if (PatternOptionBuilder.NUMBER_VALUE == c)
24       return createNumber(s);
25     // 6 more else if branches omitted.
26     else return null; } } // 7 more methods omitted.
```

Fig. 3. Two classes from *Apache CLI*. Branch coverage requires both domain knowledge on constant values and accurate type management.

```
package org.apache.commons.compress.utils;
public final class IOUtils {
  private IOUtils() { }
  public static long copy(final InputStream input,
    final OutputStream output) throws IOException {
    return copy(input, output, 8024);
  }
  public static long skip(InputStream input, long n)
    long available = n;
    while (n > 0) {
      long skipped = input.skip(n);
      if (skipped == 0) break;
      n -= skipped;
    }
    return available - n; } } // 5 methods omitted.
```

Fig. 4. Methods in *Compress* require inputs outside the fixed method pool.

the type cast is performed on o to `Class` explicitly before using it as the input of `createVal`. Without the exact type management, many branches (e. g., line 20, 24) in the method `createVal` cannot be covered, although instances that are able to cover these branches do exist.

GRT stores all successfully executed method sequences in its object pools (see Fig. 1), which can return objects as further inputs to test MUTs. To improve the effectiveness of input object selection, we manage the objects using their run-time types. This increases the type diversity of generated objects (method sequences), and thus the coverage of methods that depend on the exact type of their inputs.

When outputting the generated sequences as test cases, we compare the static type of each method return value with its dynamic type, adding explicit type casts where necessary. Otherwise, the generated tests may fail to compile, because the static types of method parameters (including the receiver) do not match the dynamic types of the objects passed to them.

The dynamic type management identifies many diverse data types and never forgets; we therefore call it *elephant brain*.

```
DemandDrivenInputCreation(T)
Input: The type T of an object to create.
Output: A set of generated objects (method sequences) of type T.
1: dependentMethodSet M ← ExtractDependentMethods(T, {})
2: for each method m ∈ M do
3:     seq ← getInputAndGenSeq(mainObjPool, secondObjPool, m)
           ▷ Get inputs for method m, and generate new method sequences
4:     if seq! = null then
5:         execSuccess ← exec(seq)        ▷ Execute method sequence seq
6:         if execSuccess then
7:             secondObjPool.add(seq)
8:         end if
9:     end if
10: end for
11: candidateMethodSeqs ← getMethodSeq(secondObjPool, T)
12: mainObjPool.addAll(candidateMethodSeqs)
13: return candidateMethodSeqs

14: ExtractDependentMethods(T, processedSet)
    Input: A dependent type T, and processedSet are types we have
    performed method extraction on.
    Output: A set of methods that constructs objects of type T .
15: M ← {}    ▷ Set of dependent methods that construct objects of type T
16: DepTypes ← {}              ▷ Set of dependent types of methods in T
17: if T ∈ processedSet ∨ T is primitive type then
        return M
18: end if
19: for each visible method m in class T do
20:     if isConstructor(m) ∨ getReturnType(m) == T then
21:         M ← M ∪ m
22:         DepTypes ← DepTypes ∪ getInputTypes(m)
23:     end if
24: end for
25: processedSet ← processedSet ∪ T
26: for each visible type T′ ∈ DepTypes do
27:     M ← M ∪ ExtractDependentMethods(T′, processedSet)
28: end for
29: return M
```

Fig. 5. Demand-driven object creation algorithm for missed input objects.

D. Detective: Demand-Driven Input Construction

To test an MUT m, all input arguments (including the receiver object) of m must be prepared. If any input of m cannot be created, m cannot be tested. Therefore, the ability of creating objects that MUTs depend on greatly affects the number of testable MUTs. In order to create auxiliary objects, diverse API types and methods are often required.

Consider class `IOUtils` (see Fig. 4), where both methods `copy` and `skip` require an object of type `InputStream`. The required object cannot be generated by tools like Randoop, because the creation of the object of type `InputStream` requires an external library (the Java core library) and cannot be performed by using only the methods in the SUT. As a result, no method in class `IOUtils` is ever covered. It is tempting to use accessible methods from dependent classes (such as all library classes) of an SUT, but this increases the search space and wastes effort on methods that are not the target.

We propose a demand-driven approach to construct missing input objects in two phases: we statically analyze the method type dependency of MUTs to identify those types that cannot be created by running MUTs only; at run-time, we use a demand-driven approach to construct inputs of types that are not directly available, by maintaining a secondary object pool.

Our method type dependency analysis first statically computes dependencies of MUTs by checking their input and return types. Then, it analyzes each input type of MUTs and determines if the objects of an input type can be obtained at run-time from other MUTs. Using this analysis, GRT identifies a set of unavailable types as candidates (input) for demand-driven input construction.

Fig. 5 shows the demand-driven algorithm for creating sequences for missing input types. When an unavailable input of type T is required during test generation, the algorithm calls function $ExtractDependentMethods$ (line 1) to search all available packages for constructors and methods that return the required type (lines 19 to 24). T is marked as processed when we have extracted the necessary methods from it. The algorithm recursively searches for inputs needed to execute a method m that returns the sought-after type T (lines 26 to 28). The recursive search terminates if the current T is a primitive type or if it has already been processed (lines 17 to 18).

For each method m required to produce objects of type T, GRT searches for necessary inputs of m in both the main and the secondary object pool. If all inputs of m are available, GRT combines the corresponding method sequences with m to generate a new method sequence ending with a call to m (lines 2 to 3). Then, GRT executes the newly created sequence and stores the resultant object in the secondary object pool (lines 4 to 9). We use a secondary object pool, because adding all objects to the main object pool can add additional overhead and decrease the query performance for the main test generation procedure. GRT selects the method sequences that produce objects of type T from the secondary object pool and adds them to the main object pool for future use (lines 11 to 12). This makes constructing missed input objects and querying efficient without interfering with the main test generation procedure.

Like a *detective*, this component works by following the clues (i. e., relationships) between methods.

E. Orienteering: Cost-Guided Input Sequence Selection

To test a method m, GRT prepares all its input objects mostly by selecting existing method sequences from the object pools. As there are often a large number of method sequences that return the objects of the same type, randomly selecting type-compatible method sequences as input makes the generated test method sequence grow considerably in length and execution cost. Even worse, repeatedly executing lengthy sequences may take up too much execution budget, leaving many other relevant sequence combinations untested.

For better run-time performance, it is desirable to use method sequences that have lower execution cost as input. The idea is inspired by *orienteering*, where a path that takes lower cost is preferable. Therefore, we randomly select a sequence as an input based on its execution cost measured by: $\text{weight}(seq) = 1/(\sum_{i=1}^{k} seq.exec\_time_i * \sqrt{seq.meth\_size})$, where seq is a sequence for selection, k counts how many times seq has been selected so far, $seq.exec\_time_i$ is the execution time during the ith execution of seq, and $seq.meth\_size$ is the number of methods in seq, excluding statements for the assignment of primitive values. This weight formula favors sequences with less execution effort while it still includes high-cost sequences with diverse states.

F. Bloodhound: Coverage-Guided Method Selection

The difficulty of covering a branch varies between branches. Some branches can be easily covered with simple inputs, while others require complex object states. An equally balanced selection of MUTs wastes time on methods that are already well covered. On the other hand, too much emphasis on MUTs containing uncovered branches may waste time in challenging the difficult branches without much payoff.

To direct testing towards uncovered code, we perform a coverage analysis during test generation and favor those MUTs that are not well covered so far. Although it is desirable to update the coverage information after each execution of an MUT, this is expensive; Therefore, the coverage information is updated at time interval t. During each interval, we prioritize method selection for a method m among all MUTs M by using the following function to compute its weight $w(m, k)$:

$$w(m,k) = \begin{cases} \alpha * \text{uncovRatio}(m) + (1-\alpha) * \left(1 - \dfrac{\text{succ}(m)}{\text{maxSucc}(M)}\right) & \text{if } k = 0 \\ \max\left(\left(\dfrac{-3}{\ln(1-p)} * \dfrac{p^k}{k}\right), \dfrac{1}{\ln(\text{size}(M)+3)}\right) * w(m,0) & \text{if } k \geq 1 \end{cases}$$

In this function, k represents the number of selections of method m since the last update of the coverage information; $\text{uncovRatio}(m)$ is the uncovered branch ratio (the number of uncovered branches over all branches) of m; p is the parameter of a logarithmic series that determines how fast the factor decreases as k increases; $\text{succ}(m)$ is the total number of successful invocations of m; $\text{maxSucc}(M)$ is the maximal number of successful invocations of all MUTs; $\max(a, b)$ returns the larger value of the two given values; $\text{size}(M)$ is the number of MUTs M; and α is the parameter to adjust the weight of the first formula.

The overall effect of the weight function is that initially ($k = 0$) we favor those methods with low code coverage. Once a method has been tested successfully ($k \geq 1$), we downgrade its weight logarithmically (the first part of $\max$ function). After several rounds of selection, the weight of each method returns to a uniform distribution again (the second part of $\max$ function). At each update of the coverage, the weights are recalculated, and k is reset to 0.

Our method selection strategy is inspired by the multi-armed bandit algorithm [61]. This algorithm balances "exploitation" (methods that are well tested) and "exploration" (methods with low coverage) for a higher payoff. The algorithm is useful because some branches of an MUT can be difficult to cover even if the MUT is tested over and over again. A weight function only based on the uncovered ratio of code would waste resources on methods with difficult branch conditions, without gaining much benefit. Our approach considers both code coverage and the execution history of each MUT for the initial weight, but decreases this weight later to avoid investing too much effort in difficult branches.

Like a *Bloodhound*, this enhancement hungers for coverage, while intelligently balancing the deeper search of each MUT against the breadth given by the entire problem set.

IV. Experiments

We implement GRT based on the random testing framework of Randoop. Constant mining is implemented as an abstract interpreter using ASM [7]. Impurity is based on ReIm & ReImInfer [26]. Bloodhound is implemented by adapting JaCoCo [28] to support on-line coverage collection during test generation. Based on our experience of developing GRT, we empirically set its parameters for the experiments; further parameter tuning is possible. For constant mining, we set the probability as $p_{const} = 0.01$. For primitive value fuzzing, we select $\sigma = 30$ as the standard deviation for Gaussian distribution fuzzing; this covers boundary conditions and character constant ranges well. For coverage guidance, we set parameters of the weight formula and time interval as $p = 0.99, \alpha = 0.9, t = 50$ seconds (see Section III-F). Using this configuration we evaluate GRT by investigating the following questions:

TABLE II. Benchmarks: Size and complexity metrics.

Software (version)	NCLOC	# Class	# Insn.	# Bran.	# Mut.
A4J (1.0b)	3,602	45	9,773	544	936
Apache BCEL (5.2)	23,631	338	65,719	5,133	7,209
Apache C. Codec (1.9)	5,803	76	24,960	1,835	2,747
Apache C. Collection (4.0)	23,713	390	47,324	5,499	7,401
Apache C. Compress (1.8)	17,462	181	57,083	4,634	7,605
Apache C. Lang (3.0)	18,997	141	47,773	7,179	9,057
Apache C. Math (3.2)	81,792	845	288,250	18,576	41,023
Apache C. Primitive (1.0)	9,836	231	18,462	1,446	3,290
Apache Commons Cli (1.2)	1,978	20	3,588	490	512
Apache Shiro-core (1.2.3)	13,818	217	27,964	3,291	3,770
ASM (5.0.1)	24,193	176	65,146	7,475	9,765
ClassViewer (5.0.5b)	1,485	23	5,266	470	609
Dcparseargs (10/2008)	204	6	652	88	103
Easymock (3.2)	4,372	79	9,449	915	1,382
Fixsuite (R48)	2,665	36	6,520	374	804
Guava (16.0.1)	66,566	1,546	136,321	11,247	20,709
Hamcrest-core (1.3)	1,253	40	2,199	155	314
Jcommander (1.36)	2,154	34	5,688	640	686
Java Simp. Arg. Parser (2.1)	4,888	69	8,623	714	969
Java View Control (1.1)	4,617	24	15,650	2,064	2,084
Javassit (3.19)	34,574	367	87,381	8,830	n. a.
Javax Mail (1.5.1)	28,271	284	79,599	9,523	11,070
Jaxen (1.1.6)	20,345	175	20,352	3,323	4,338
Jdom (1.0)	8,362	70	20,970	3,196	4,116
Joda Time (2.3)	27,638	208	62,627	6,172	9,838
Mango (2.1 03/2014)	2,141	90	3,689	382	556
Nekomud (R16)	363	8	809	44	63
Pmd-dcd (5.2.2)	1,608	20	2,902	305	384
SAT4J Core (2.3.5)	17,397	213	41,840	3,815	6,140
SCCH collection (1.0)	1,348	25	2,688	292	433
Slf4j-api (1.7.12)	1,504	18	2,581	271	265
Tiny Sql (2.26)	7,672	31	20,850	2,237	2,755
Total	464,252	6,026	1,192,698	111,159	160,933

Q1: What code coverage and mutation score are achieved by GRT, compared to Randoop and EvoSuite?
Q2: How does each tool perform given different time budgets?
Q3: How much does each component of GRT contribute to code coverage?
Q4: How many existing defects can be detected by GRT in a controlled study?
Q5: How many new defects can GRT reveal in real-world software?

A. Subject Programs and Setting

We compare GRT with Randoop 1.3.4 [49], and with EvoSuite (snapshot Oct. 14, 2014) [14]. We select EvoSuite because it represents the state of the art in search-based testing [21], [19].

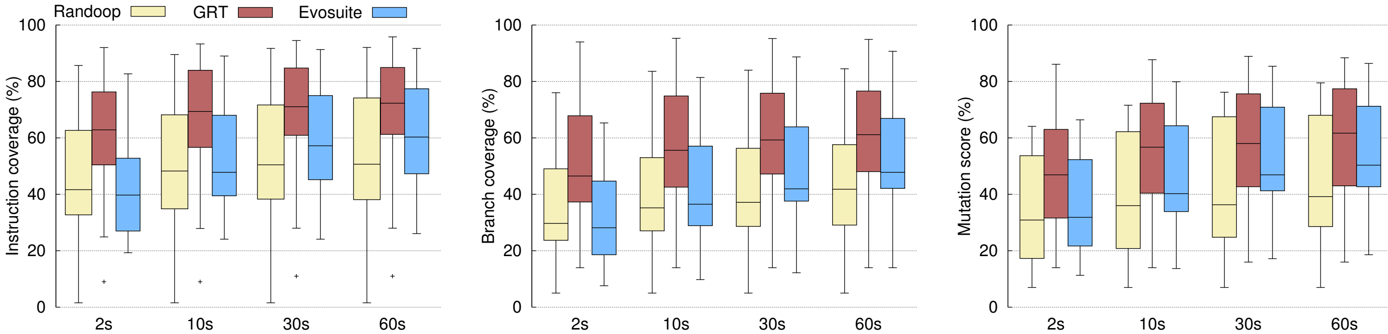

Fig. 6. Instruction coverage, branch coverage and mutation score of Randoop, GRT and EvoSuite over 32 subjects for a time budget of 2 s to 60 s/class.

TABLE III. RESULTS: AVERAGE INSTRUCTION/BRANCH COVERAGE AND MUTATION SCORE OVER 32 BENCHMARKS, FOR DIFFERENT TIME BUDGETS.

Time budget	Insn. cov. [%]			Branch cov. [%]			Mutation score [%]		
	Ran.	GRT	Evo.	Ran.	GRT	Evo.	Ran.	GRT	Evo.
2 s	47.4	60.6	43.8	35.1	49.5	32.6	34.5	47.3	36.8
10 s	51.1	66.3	52.0	39.3	56.7	42.5	39.3	54.7	46.1
30 s	52.9	68.2	57.8	41.3	59.2	49.4	41.9	57.8	51.9
60 s	53.6	68.9	60.8	42.6	60.3	53.3	43.9	59.2	54.5

To answer Q1–Q3, we run all tools on a collection of 32 popular real-world programs. The overview in Table II shows for each program its name and version, its overall size in terms of non-comment lines of source code (NCLOC, measured by CLOC 1.60 [11]), the number of classes, the number of instructions and branches in the bytecode (measured by JaCoCo v0.6.4 [28]), and the number of mutants generated by the mutation analysis tool PIT [47].

Our experiments were executed on a computer cluster. Each cluster node ran a GNU/Linux system (Ubuntu 12.04 LTS) with Linux kernel 3.5.0, on a 16-core 1.4 GHz AMD 64-bit CPU with 48 GB of RAM. We used Oracle's Java VM (JVM) version 1.7.0_65, allocating up to 4 GB for the JVM.

B. Code Coverage and Mutation Score

Q1 and Q2: We compare the effectiveness of GRT, Randoop and EvoSuite, in terms of code coverage and mutation score. We run each tool on each study subject with four test time budgets: 2 s/class, 10 s/class, 30 s/class, and 60 s/class. Pre- and post-processing, such as loading classes and writing test cases to disk, are not counted towards that time budget. As also discussed in other work [17], we use different time budget configurations to account for different use cases, from testing during a coffee break to generating tests over night. For each configuration (time budget, tool, subject), the experiments are repeated 10 times to mitigate the influence of the randomness of the tools. As each tool sometimes generates tests that do not compile, our experimental platform automatically removes uncompilable code at a method level. All the compilable tests are then evaluated by JaCoCo for code coverage. As a conventional procedure for mutation analysis [30], we first filter out generated test cases that fail on the original programs, and then send the passing tests to PIT to compute the mutation score that measures the ability of killing automatically generated mutants.

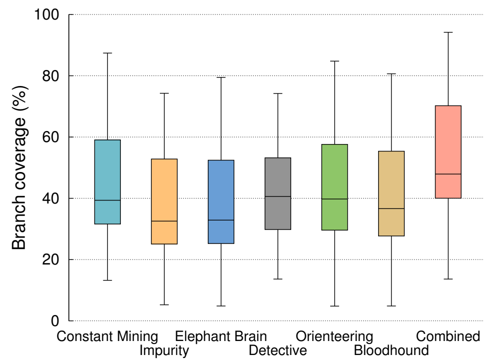

Fig. 7. Branch coverage for each component on our 32 benchmarks (600 s).

With a time budget of 60 s/class, the tools mostly reach a state in which code coverage and mutation score grow much slower or stop growing (with few exceptions when running EvoSuite). Since the amount of time is allocated for *each class* (instead of the entire SUT), the results are largely independent of the size of the subject programs. When extending the time budget, large subjects (e.g., Guava) run for many hours, as mutation analysis incurs a high computational cost [30] and sometimes takes longer than test generation itself. For example, on Jaxen, each tool takes about 2 hours to finish test generation when choosing 60 s/class as time budget, but it takes more than 10 hours for running the mutation analysis with PIT. In total, over all cluster nodes, the experiments consumed more than one year of computation time.

Figure 6 shows instruction coverage, branch coverage and mutation scores achieved by Randoop, GRT and EvoSuite over the 32 study subjects for each time budget configuration ranging from 2–60 s/class. Table III summarizes the results in terms of average code coverage and mutation score. The results show that when running with a short time budget, (2 s or 10 s), GRT has a clear advantage on both higher code coverage (by 28–52 %) and mutation score (by 19–39 %) compared with Randoop and EvoSuite. When the provided test time budget increases, the overall code coverage and mutation score of all tools increase, too. EvoSuite shows a noticeable improvement from 2 s to 60 s, reducing the coverage gap between GRT and EvoSuite. This is because EvoSuite first performs an initial random search and then uses evolutionary search to improve its results [19]; the latter phase requires a certain amount of time to become effective. With the largest time budget of 60 s/class the coverage of EvoSuite tends to plateau out. For Randoop, coverage tends to saturate after about 30 s/class.

For the largest time budget of 60 s/class, the average branch coverage from GRT is 42 % $= (60.3 - 42.6)/42.6$ higher than with Randoop and 13 % higher than with EvoSuite. For the average instruction coverage, the values are 29 % and 13 %, respectively. On the average mutation score, GRT also outperforms Randoop by 35 % and EvoSuite by 9 %, indicating that the tests generated by GRT have better performance in revealing automatically seeded faults (mutants). The evaluation of the coverage and mutation scores over all 32 subjects shows that the improvement of GRT over Randoop and EvoSuite is statistically significant (Wilcoxon Matched-Pairs Signed-Ranks Test [53], $p < 0.05$ in all cases). The effect size is determined using Vargha and Delaneys A measure [60]; A = 0.58 to 0.73. For assessing the results of the individual subjects we follow the guidelines proposed by Arcuri and Briand [3]. The results including code coverage and mutation scores for each benchmark are available on our website [24].

These results were also confirmed by the Search-Based Software Testing Competition [50], [36], where GRT competed with six other tools, also including Randoop and Evosuite. The tools were compared over a benchmark that was not revealed to participants a priori, following a fully automated competition protocol that evaluated the effectiveness and efficiency of the tools. The benchmark contained 63 classes taken from 10 open source Java packages. GRT achieved the highest score of all tools [50], which was calculated based on obtained code coverage, mutation score, and the time used to prepare, generate and execute the test cases [50].

Q3: We run GRT with each of its six components enabled individually in comparison to GRT with all components enabled on our 32 subjects with 600 s as global time budget. We observe a coverage improvement for each component and for full GRT as time increases. In general, each individual component of GRT contributes to the overall effectiveness; the impact of each component varies across different subjects. The combination of all six components is usually stronger than any single component, as can be seen from the branch coverage boxplots over all 32 subjects (Fig. 7).

Fig. 8–Fig. 10 show three examples of how each component improves code coverage. Constant mining is effective when extracted constants relate to branch conditions (Fig. 8). Sometimes, detective makes a breakthrough by automatically constructing objects of specific types (Fig. 9). Fig. 10 shows another example, where orienteering outperforms the other components of GRT. Other plots can be found online [24].

C. Defect Detection

Q4 and Q5: We first evaluate the defect detection ability of each tool on the Defects4J framework [31], and then use GRT to find new unknown defects in popular open source projects.

1) Defects4J: The Defects4J framework enables testing studies using existing real faults. It contains 357 real faults reported in five open source projects [31], [32]. For each fault, Defects4J uses two versions of the program: a faulty version and a correct version. Defects4J first runs a testing tool on the correct version to generate test suites, and then runs the generated test suites on the faulty version to see if the bug is detected.[1] The outcome of a technique on a specific fault is *Pass*, *Fail*, or *Broken*, which means the fault is not detected, successfully detected, or the tests fail on the bug-free version (in this case, the generated tests cannot be used to determine whether they could detect bugs), respectively. To make experiments efficient, Defects4J provides the information on fault-related classes, so that tools can focus on these classes (instead of the entire SUT) when generating test cases.

We run GRT, Randoop, and EvoSuite on Defects4J to compare their fault-detection ability in a controlled environment (i. e., the faults are known). We use 120 seconds, 300 seconds and 600 seconds as the *global* time budget when comparing GRT and Randoop in different use cases, and allocate 120 seconds, 300 seconds and 600 seconds for *each class* when running EvoSuite on Defects4J. This should be sufficient for each tool to generate test cases and is reasonable for our available computing resources. To mitigate the effects of randomness, we run each tool 10 times to generate 10 test suites (one test suite each time) to detect each fault. We then measure the faults detected by each tool by aggregating the faults found by the 10 generated test suites in each setting.

As shown in Table IV, each tool detects more bugs when using a larger time budget setting, and GRT shows the largest improvement ($29 = 147 - 118$) when the time budget increases from 120 s to 600 s. GRT also detects more faults than Randoop and EvoSuite in all subjects. In particular, GRT can detect 23 out of 26 (88 %) faults in JFreeChart, 21 out of 27 (77 %) faults in JodaTime, and more than 50 % of the real faults in both Apache Math and Apache Lang using 600 s as global time budget. This result demonstrates GRT's strong fault detection ability, in a controlled study using a large number of real faults under different time budget settings.

Table IV shows the four cases where we were able to replicate most of the data on Randoop and EvoSuite from a previous study on Defects4J [32]. Table IV does not include data for the fifth subject, Closure Compiler, because all tools detect unexpectedly few faults.[2] Other minor deviations from the previous study can be attributed to differences in our computing environment, including hardware and software, and the exact configurations; we used mostly default settings.[3]

2) Open Source Software: To evaluate GRT's ability to find new, previously unknown defects, we apply it to the latest versions of 10 popular, widely used open source projects. We use system exceptions, crashes, and the behaviors stipulated for the base class `java.lang.Object` (e. g., the reflexivity property of `equals()`) as the test oracle [43].

As the failed tests generated by GRT require manual analysis to determine whether they reveal *true* bugs or generate false positives, we selected projects that are still under active

[1] Defects4J removes all uncompilable tests, failed tests, and non-deterministic tests before running a test suite for bug detection.

[2] In private communications with an author of Defects4J, we confirm that it is quite challenging to generate useful test cases for Closure Compiler, but we could not confirm the root causes; this deserves future research.

[3] In their study on correlations between mutants and real faults [32], the authors of Defects4J generated 30 test suites with EvoSuite (per configuration) and 6 test suites with Randoop (per configuration) for each subject. As our aim is to compare different techniques, we generate the same number of test suites (i. e., 10) with Randoop, GRT, and EvoSuite with similar time budget on each setting, to make the comparison as fair as possible.

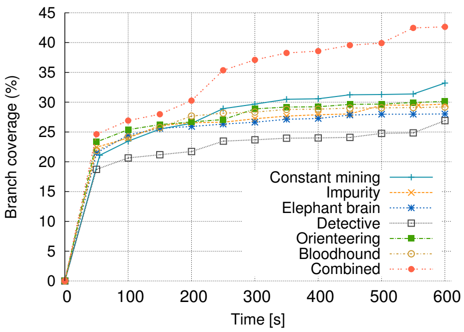

Fig. 8. Branch coverage over time (Tiny sql).

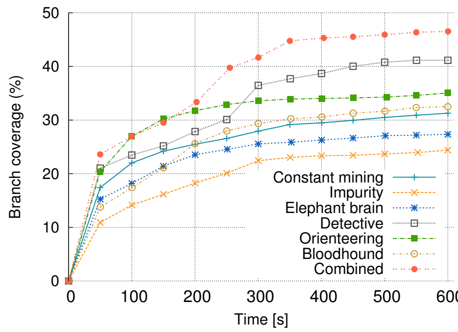

Fig. 9. Branch coverage by time (ASM).

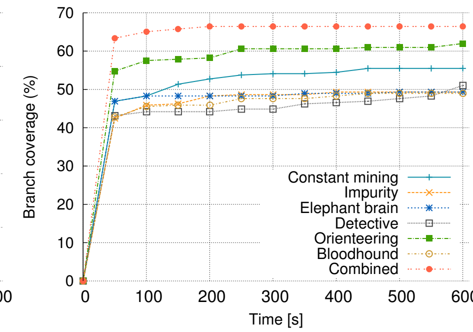

Fig. 10. Branch coverage by time (SCCH coll.).

TABLE IV. DEFECT DETECTION IN DEFECTS4J BENCHMARKS

Program	Isolated defects	Ran. (global time)			GRT (global time)			Evo. (time/class)		
		120 s	300 s	600 s	120 s	300 s	600 s	120 s	300 s	600 s
JFreeChart	26	15	15	17	20	23	23	15	17	18
Apa. Math	106	32	39	45	58	66	68	48	53	58
Joda-Time	27	12	12	12	12	19	21	13	15	16
Apa. Lang	65	10	14	14	28	30	35	21	24	27
Total	224	69	80	88	118	138	147	97	109	119

TABLE V. DEFECT DETECTION: GRT FINDS 23 PREVIOUSLY UNKNOWN DEFECTS.

Software	Failed tests	Filtered tests	Identified issues				Issue numbers
			issues	false	unkn.	true	
A. CLI	1	1	0	0	0	0	–
A. Codec	2	1	1	0	0	1	183, 184
A. Collection	56	25	15	13	0	2	512–516
A. Compress	25	0	4	0	0	4	273–276
A. Math	76	45	13	6	0	7	1115–1118, 1224
A. Primitive	4	0	2	1	1	0	17
Guava	13	5	8	6	0	2	1722–1724
JavaMail	20	12	6	0	0	6	6365–6368
Mango	3	1	1	0	0	1	1
TinySQL	8	0	6	1	5	0	14–18
Total	208	90	56	27	6	23	

development (the last update being less than a year ago), and for which the number of failed test cases is not prohibitively high (i. e, fewer than 100 failed tests); see Table V.

From the 208 failed tests, we first filter out tests that confirm a problem that is either known or not going to be fixed in the code, such as bugs caused by using deprecated methods and infinite recursion in container data structures. We then manually simplify the remaining tests and identify duplicates by comparing the stack traces and the sequences of method calls of different tests. This results in 56 distinct issues. We reported these using the projects' bug tracking systems, combining similar issues into one bug report. According to the developers' feedback, GRT found **23** new, previously unknown defects (see Table V).

D. Summary

Compared with Randoop and EvoSuite, GRT significantly improves code coverage and mutation scores (Q1). The advantage of GRT is observed for all time budget configurations, from 2 s/class to 60 s/class; the tools mostly tend to reach a plateau at 60 s/class. Compared with Randoop and EvoSuite, GRT achieves a high coverage sooner (Q2). Not all components of GRT are equally effective in all cases, yet the overall effectiveness of GRT results from the synergy between all six components (Q3). GRT is able to detect about two thirds of the known faults on the studied subjects of Defects4J (Q4) as well as a number of new, previously unknown faults in the latest versions of real-world programs (Q5).

E. Threats to Validity

The selection of study subjects is always a threat to validity. We try to counter this by choosing 32 diverse programs from various application domains with their sizes ranging from very small to fairly large. An external threat to validity is caused by the randomness of the three tools. We run each tool on the same configuration 10 times to diminish this threat, and have not observed significant variance caused by randomness. A related threat is that different tools may require different amounts of time to exhibit their best performance. As a countermeasure, we use four different time budgets to study the effectiveness of each tool in most typical use cases. We have fully utilized our computing resources to extend the time budget as much as possible (up to 60 s/class). Another threat is that we have not examined all tool configurations. In particular, EvoSuite can be configured to satisfy one of three criteria, including branch coverage, weak mutation testing, or strong mutation testing. Our study uses the default configuration, which is branch coverage. However, as indicated in a previous study [32] on Defects4J, the other two configurations would yield similar overall results in terms of detected bugs. From the authors of Defects4J, we also obtained the breakdown of their earlier study [32] and confirmed that there are minor differences between results generated by different configurations.

We did not compare GRT with test generation tools based on symbolic execution. This may miss an important aspect of our study. It is because we could not find an existing symbolic execution based automatic test tool that supports to test Java programs and works on the large set of subjects that we used. However, the idea of symbolic execution is orthogonal to the framework of GRT and could be integrated as another analysis component in GRT in the future.

V. RELATED WORK

Given the large body of work on automated testing, we discuss only work closely related to GRT. For further work, we refer readers to representative surveys [1], [39], [46], [18].

1) Variants of Random Testing: The critical step in automatic test case generation for object-oriented programs is to prepare input objects with desirable object states. An input object can be constructed by either direct construction [6], [38] or method sequence construction returning the desired objects [44], [55], [66]. Direct construction approaches, e. g., Korat [6] and TestEra [38], construct objects by assigning fields directly. They use specifications defined in languages, such as Alloy, and are therefore not fully automated.

Most random techniques create required input objects by *method sequence construction* [12], [42], [44], [55], [65]. JCrasher [12] creates input objects by using a parameter graph to find method type dependencies (similar to our dependency method extraction described in Section III-D). Eclat [42] and Randoop [44], [43] use feedback from previous tests. The run-time phase of GRT is based on the same basic idea, however, it performs sophisticated dynamic analysis to generate finer-grained feedback. In addition, the static phase of GRT extracts useful information of the SUT to support the run-time phase.

Adaptive random testing (ART) [8], [1] improves the defect detection effectiveness of random testing by evenly spreading test input selection across the input domain. Since its introduction by Chen et al. [9], various studies [8], [10], [33] have shown that ART requires fewer tests to detect defects than random testing. However, it has also been shown that ART has a high computational overhead [2], [45], and has difficulties in testing large SUTs that require complex inputs [1]. It would be interesting to include adaptive random testing (ART) tools in the analysis of GRT as well, as our constant mining technique is related to it. Unfortunately, we are not aware of any publicly available ART tools that support Java and work on the large set of benchmarks we used. We leave the study on the usefulness of ART as a GRT component as future work.

2) Random Testing Guided by Domain Knowledge: Several tools take advantage of the information contained in existing test cases (method sequences). MSeqGen [55] mines frequently used sequence patterns from code bases. Palus [65] trains a method sequence model from existing test cases, which is used for test generation at run-time. OCAT [29] adopts object capture-and-replay techniques, where object states are captured from running sample test cases and then used as input for further testing. Similar to these techniques, GRT also makes use of program analysis to guide random testing, but GRT does not require extra information sources, such as existing test cases and code bases.

3) Systematic Testing: In contrast to random testing, *symbolic execution* represents input as symbolic values, execution is based on abstract semantics, and path conditions are computed by leveraging constraint solvers. Tools like Java PathFinder [62] and Symbolic PathFinder [34] generate test cases in this way. Hybrid approaches of random (concrete) and symbolic execution, called *concolic* execution, are implemented by tools like DART [23], Cute and JCute [52], [51], Pex [56], and Dsc [27].

An alternative to symbolic execution is *bounded exhaustive testing* [38], [6], [63], which exhaustively generates method sequences up to a small bound of sequence length. However, real-world software usually requires longer test sequences to examine more program states beyond a small bound.

4) Evolutionary Testing: Evolutionary testing [57], [4], [19], [16] leverages evolutionary algorithms to evolve and search for test sequences that optimize their fitness, e. g., branch coverage, in a limited search budget. EvoSuite [19], [16], [22] implements such an approach. Yet it goes beyond traditional techniques as it adopts a hybrid approach to automatically generate and optimize the whole test suites towards satisfying coverage criteria. It has been shown effective in achieving high coverage on real-world software [18], [20]. GRT shares some ideas with EvoSuite, such as extracting constants from SUT. Using EvoSuite, Fraser and Arcuri [17] study the influence of seeding constants (extracted from SUT) on the search-based testing techniques. The constant mining component of GRT is based on a similar assumption: Constants used in the SUT are more likely to be useful in testing. However, GRT uses different strategies, namely frequency-based prioritization and value fuzzing, to improve the usefulness of the extracted constants. We have not investigated how constants extracted from existing test cases can improve GRT (the third strategy studied by EvoSuite [17]). Although this can be a promising enhancement to GRT, it would make GRT dependent on external knowledge (i. e., existing tests).

VI. Conclusion and Future Work

In this paper, we propose GRT, a technique that combines static analysis and run-time analysis to guide random testing. GRT does not rely on knowledge outside of the SUT. Our static analysis extracts domain knowledge from the SUT as input for run-time test generation. Our dynamic analysis systematically improves test coverage in the generation phase. We have evaluated GRT thoroughly on a large set of real-world projects. Our approach exhibits significant improvements on code coverage, mutation score, and the ability to find defects.

Our work shows that random testing has not reached its limits yet. GRT itself can be improved in a number of ways. It is tempting to incorporate symbolic execution techniques to achieve higher code coverage, especially in the face of complicated branches. Simple specialized treatments, such as handling less visible code, may be surprisingly effective. We also plan to enhance the test oracle of GRT. Currently, GRT focuses on leveraging program analysis to obtain high code coverage, using simple oracles, such as software crashes and exceptions. Sometimes the oracles are too weak to detect the faults, even though the faulty code is executed. Automated specification mining that extracts information on valid uses of a system [13], [48] would be a promising next step towards stronger test oracles. Considering the sheer number of generated test cases, reducing false positives is another important task. Possible solutions include options to avoid deprecated code and recursive data structures. Developing an efficient test simplification technique is also helpful to ease the validation of failed tests. Enabling the application of GRT in more scenarios [35] is another direction of our future work.

VII. Acknowledgments

We thank Reid Holmes, Mauro Pezzè, and Sai Zhang for their insightful comments, and Gordon Fraser, José Campos and René Just for their help on Evosuite and Defects4J. This work was supported by the *SEUT* project from the University of Tokyo and *kaken-hi* grants 23240003 and 26280019.

REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.

[2] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, pages 265–275, Toronto, Ontario, Canada, 2011.

[3] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 1–10, Waikiki, Honolulu, HI, USA, 2011.

[4] L. Baresi and M. Miraz. Testful: Automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE'10, pages 281–284, Cape Town, South Africa, 2010.

[5] S. Bauersfeld, T. Vos, and K. Lakhotia. Unit testing tool competitions lessons learned. In *Future Internet Testing*, pages 75–94. 2013.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'02, pages 123–133, Roma, Italy, 2002.

[7] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[8] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proceedings of the 9th Asian Computing Science Conference on Advances in Computer Science: Dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday*, ASIAN'04, pages 320–329, Chiang Mai, Thailand, 2004.

[9] T. Y. Chen, T. H. Tse, and Y. T. Yu. Proportional sampling strategy: A compendium and some insights. *J. Syst. Softw.*, 58(1):65–81, Aug. 2001.

[10] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 71–80, Leipzig, Germany, 2008.

[11] CLOC 1.60. http://cloc.sourceforge.net/.

[12] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[13] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering*, 38(2):243–257, March 2012.

[14] Evosuite-20141014. http://www.evosuite.org/downloads/.

[15] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 59–68, Seattle, Washington, 2000.

[16] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE'11, pages 416–419, Szeged, Hungary, 2011.

[17] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, ICST'12, pages 121–130, Los Alamitos, CA, USA, 2012.

[18] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 178–188, Zurich, Switzerland, 2012. IEEE Press.

[19] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, Feb. 2013.

[20] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[21] J. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Proceedings of the 24th International Symposium on Software Reliability Engineering*, ISSRE'13, pages 360–369, 2013.

[22] J. P. Galeotti, G. Fraser, and A. Arcuri. Extending a search-based test generator with adaptive dynamic symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 421–424, San Jose, CA, USA, 2014.

[23] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[24] GRT Additional Result. https://sites.google.com/site/grtprojectut/ase15.

[25] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[26] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & ReImInfer: Checking and inference of reference immutability and method purity. *SIGPLAN Not.*, 47(10):879–896, Oct. 2012.

[27] M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *Proceedings of the 8th International Workshop on Dynamic Analysis*, WODA'10, pages 26–31, Trento, Italy, 2010.

[28] JaCoCo v0.6.4. http://www.eclemma.org/jacoco/.

[29] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: Object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA'10, pages 159–170, Trento, Italy, 2010.

[30] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

[31] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 437–440, San Jose, CA, USA, 2014.

[32] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 654–665, Hong Kong, China, 2014.

[33] Y. Lin, X. Tang, Y. Chen, and J. Zhao. A divergence-oriented approach to adaptive random testing of java programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE'09, pages 221–232, Auckland, New Zealand, 2009.

[34] K. S. Luckow and C. S. Păsăreanu. Symbolic pathfinder v7. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[35] L. Ma, C. Artho, C. Zhang, and H. Sato. Efficient testing of software product lines via centralization (short paper). In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE'14, pages 49–52, Vasteras, Sweden, 2014.

[36] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe, and M. Yamamoto. GRT at the SBST 2015 tool competition. In *The 8th Int. Workshop on Search-Based Softare Testing*, SBST'15, pages 48–51. Florence, Italy, 2015.

[37] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[38] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE'01, pages 22–31, San Diego, CA, USA, 2001.

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

[40] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[41] C. Oriat. Jartege: A tool for random generation of unit tests for java classes. In *Proceedings of the 1st International Conference on Quality of Software Architectures and Software Quality, and Proceedings of the 2nd International Conference on Software Quality*, QoSA'05, Erfurt, Germany, 2005.

[42] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Glasgow, UK, 2005.

[43] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA'08, pages 87–96, Seattle, WA, USA, 2008.

[44] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 75–84, Minneapolis, MN, USA, 2007.

[45] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE'12, pages 35:1–35:11, Cary, North Carolina, 2012.

[46] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, Oct. 2009.

[47] PIT 1.1.3. http://pitest.org/.

[48] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 288–298, Zurich, Switzerland, 2012.

[49] Randoop 1.3.4. https://code.google.com/p/randoop/downloads/list.

[50] U. Rueda, T. Vos, and I. Prasetya. Unit testing tool competitions - round three. In *The 8th Int. Workshop on Search-Based Softare Testing*, SBST'15, pages 19 – 24. Florence, Italy, 2015.

[51] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Seattle, WA, USA, 2006.

[52] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, Sept. 2005.

[53] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[54] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, Paris, France, 2005.

[55] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE'09, pages 193–202, Amsterdam, The Netherlands, 2009.

[56] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Prato, Italy, 2008.

[57] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'04, pages 119–128, Boston, USA, 2004.

[58] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, pages 211–230, San Diego, CA, USA, 2005.

[59] Valentin Dallmeier, Christian Lindig, Andreas Zeller. Dynamic purity analysis for Java programs, https://www.st.cs.uni-saarland.de/models/jpure/, 2007.

[60] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[61] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of the 16th European Conference on Machine Learning*, ECML'05, pages 437–448, Porto, Portugal, 2005.

[62] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[63] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Edinburgh, UK, 2005.

[64] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE'07, pages 75–82, San Diego, California, USA, 2007.

[65] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, pages 353–363, Toronto, Ontario, Canada, 2011.

[66] W. Zheng, Q. Zhang, M. Lyu, and T. Xie. Random unit-test generation with mut-aware sequence recommendation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 293–296, Antwerp, Belgium, 2010.