Saying 'Hi!' Is Not Enough: Mining Inputs for Effective Test Generation

Luca Della Toffola
Department of Computer Science
ETH Zurich, Switzerland

Cristian-Alexandru Staicu
Department of Computer Science
TU Darmstadt, Germany

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany

***Abstract*—Automatically generating unit tests is a powerful approach to exercise complex software. Unfortunately, current techniques often fail to provide relevant input values, such as strings that bypass domain-specific sanity checks. As a result, state-of-the-art techniques are effective for generic classes, such as collections, but less successful for domain-specific software. This paper presents TestMiner, the first technique for mining a corpus of existing tests for input values to be used by test generators for effectively testing software not in the corpus. The main idea is to extract literals from thousands of tests and to adapt information retrieval techniques to find values suitable for a particular domain. Evaluating the approach with 40 Java classes from 18 different projects shows that TestMiner improves test coverage by 21% over an existing test generator. The approach can be integrated into various test generators in a straightforward way, increasing their effectiveness on previously difficult-to-test classes.**

I. Introduction

Automated test generation is a powerful approach to create inputs for exercising a software under test with minimal human effort. Existing approaches use a wide range of techniques, ranging from feedback-directed random test generation [5], [37], [39], over search-based approaches [11], [17], to symbolic reasoning-based test generators [8], [21], [45], [52]. Despite all successes, test generation still suffers from non-trivial limitations. For example, a study reports that only 19.9% of bugs in a well known collection of existing faults are revealed by the test suites generated by three state-of-the-art test generators [48].

An important limitation is that test generators often fail to cover a buggy statement because the inputs provided in the test do not enable the code to bypass sanity checks that reject invalid inputs. In particular, creating bug-revealing inputs often requires suitable strings, but creating such strings requires domain knowledge about the software under test, which existing test generators do not have. For example, consider testing a class responsible for parsing SQL statements. Testing the class with a randomly generated string is highly unlikely to reach deeply into the code because the invalid input is discarded quickly due to a parse error.

State-of-the-art test generators obtain input data, such as strings, in various ways. First, most generators use randomly generated values or values from a fixed pool, which are cheap to obtain but unlikely to match domain-specific data formats. For example, Randoop [37] often uses the value "hi!" as a string value. Second, some test generators extract constants, e.g., stored in fields of the class under test, and return values of previously called methods and use these values as inputs. This approach is effective if suitable constants and methods are available, but fails otherwise.

Third, some test generators symbolically reason about expected values [22], e.g., based on a constraint solver able to reason about strings [55]. While effective, this approach often suffers from scalability issues and may not provide the best cost-benefit ratio, which is crucial for testing [6]. Finally, some test generators rely on a grammar that describes expected string values [20], [54]. However, including grammars for all, or even most, domain-specific input formats into a general-purpose test generators is impractical.

This paper present TestMiner, a novel technique to address the problem of generating domain-specific input values suitable for a given software under test. The key idea is to exploit the wealth of information available in existing code bases, in particular in existing tests, using an information retrieval-inspired mining technique that predicts input values suitable for testing a particular method. The approach consists of two phases. At first, the approach extracts literals from the source code of existing tests and indexes them for quick retrieval. Then, a test generator queries the mined data for values suitable for a given method under test. These predicted values are then used as test inputs during test generation. Our idea can be incorporated into any automated test generator that requires input values of primitive types or strings.

In summary, this paper makes the following contributions:

- *Information Retrieval for Test Inputs*. We are the first to exploit the knowledge hidden in large amounts of existing code to address the problem of finding suitable input values for testing.
- *Scalable and Efficient Prediction of Domain-specific Values*. We present a scalable and efficient technique to predict input values suitable for a given method under test and show how to integrate the technique into an existing test generator.
- *More Effective Test Generation*. We show empirically that the presented approach positively influences the effectiveness of a state-of-the-art test generator.

This work has been supported by the DFG within ConcSys, and by the BMBF and the HMWK within CRISP, and in part by SNF grant 206021_133835.

II. Approach

This section presents TestMiner, an approach for mining test input values from existing projects. Figure 1 gives an overview of the main steps.

A. Static Analysis of Test Suites

This first step extracts input values from a corpus of tests and associates to each literal value a context to be used later for retrieving values during test generation.

Definition 1 (Context-value pair)**.** *A context-value pair $(\mathcal{C}, v)$ consists of a context $\mathcal{C} : \mathcal{S} \rightarrow \mathbb{N}$, represented as a bag of words, and a value $v \in \mathcal{V}$ in some value domain. The set $\mathcal{S}$ refers to the set of all strings.*

The context $\mathcal{C}$ may be anything that can be represented as a bag of words. There are various options for defining the context, such as the calling context for the method under test, the type hierarchy of the program, or the signature of a method that is tested. In this work, we compute the context of a value from the fully qualified signature of the method that receives the value as an argument. This notion of context works well because fully qualified method signatures often contain rich semantic information [24].

For each test in the corpus, the static analysis collects the fully qualified method signature of all calls sites that pass a literal argument, and it annotates each argument with its static type. The analysis returns for each test a set of call site tuples.

Definition 2 (Call site tuple)**.** *A call site tuple $S_c = (\mathcal{T}_c, m_c, \mathcal{V}_c)$ consists of a set $\mathcal{T}_c$ of fully qualified type names in which the method m_c is defined, the method name m_c, and a set $\mathcal{V}_c$ of values passed as arguments at the call site.*

For example, suppose a test case calls `sqlParser.parse("SELECT x FROM y")`. Then the static analysis extracts, the following call site tuple: $(\{org.sql.SQLParser\}, parse, \{"SELECT\ x\ FROM\ y"\})$. The set $\mathcal{T}_c$ may, in principle, contain multiple type names because a call may resolve to multiple methods. Our static analysis considers only the statically declared type of the base object of a call, i.e., $|\mathcal{T}_c| = 1$. Completely resolving all possible call targets would require a whole-program analysis, which does not scale to a large corpus. For the set $\mathcal{V}_c$ of values, we focus on string values in this work. Applying the idea to another value domain, e.g., integers, is straightforward. The reason is that finding suitable strings is a major obstacle for state-of-the-art test generators [48].

Finally, the analysis transforms the tuples into context-value pairs. For this purpose, the approach tokenizes the type names in $\mathcal{T}_c$ and the method name at dot-delimiters, splitting strings based on the camelCase and snake_case conventions, and it normalizes the remaining tokens into lower case. The resulting strings are then represented as a bag of words, which represents the context $\mathcal{C}$. For the above example, the analysis yields this context-value pair:

$$(\{org \mapsto 1, sql \mapsto 2, parser \mapsto 2\}, "SELECT\ x\ FROM\ y")$$

In the second step the approach summarizes and indexes context-value pairs for a later retrieval. The basic idea is to associate each input value with one or more hash values that summarize the context in which the input value occurs. The resulting hash map then serves efficiently retrieving values suitable for a particular context. These steps are summarezied in Algorithm 1 and explained in detail here.

Algorithm 1 Summarize and index context-value pairs.

Input: Set $\mathcal{P}$ of context-value pairs
Output: Index-to-values map $\mathcal{M}$

1: $\mathcal{M} \leftarrow$ empty map
2: **for all** $(\mathcal{C}, v) \in \mathcal{P}$ **do**
3: $\quad \mathcal{C}_{weighted} \leftarrow normalize(tfidf(\mathcal{C}))$
4: $\quad h \leftarrow simHash(\mathcal{C}_{weighted})$
5: $\quad$ update $\mathcal{M}(h)$ with v
6: **end for**
7: **return** $\mathcal{M}$

1) Assigning Weights to Context Words: While some words in the context convey useful information about the domain of the tested code, others may be redundant. For example the word "sql" is crucial to describe the context for a method that expects an SQL query. In contrast, words such as "org" are very frequent and occur across unrelated domains.

To enable TestMiner to focus on the most relevant words in a context, we compute a weight for each word using the *term frequency-inverse document frequency* (tfidf). which represents how important a word is to a document in a corpus of documents. This measure is commonly used for information retrieval. Formally, we compute tfidf as

$$tfidf(t, d, D) = f_{t,d} \cdot log\Big(\frac{|D|}{|\{d \in D : t \in d\}| + 1}\Big) \quad (1)$$

where document d is the context $\mathcal{C}$, a term t is a context word in $\mathcal{C}$, the corpus D is the set of all contexts, and where $f_{t,d}$ is the frequency of term t in document d. For the above example a low weight is assigned to "org" because this word occurs frequently in the corpus, and it assigns a relatively high weight to "sql" because this word is relatively uncommon but appears twice in the context. As a result, the approach champions the informative parts of the signature and penalizes the less informative ones.

2) Indexing with Locality-Sensitive Hashing: To index the context words, we use the locality-sensitive hash function Simhashing [10]. This class of hash functions is designed to assign to similar values a similar hash value with high probability, preserving the value similarity also in the hash space. In addition to efficient retrieval through hashing, this choice of hash function enables TestMiner to generalize beyond the exact contexts extracted from the corpus. After assigning weights to context words, TestMiner indexes the values for efficient retrieval (line 4) and stores values indexed by their hash value (line 5). The final result of the indexing step of TestMiner is a map that assigns hash values to test input values:

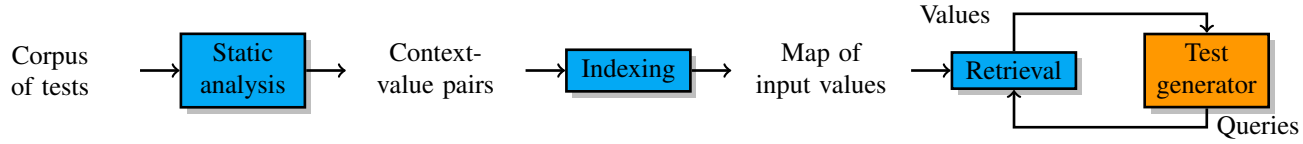

Fig. 1. Overview of the TestMiner approach. The blue components are part of the approach.

Definition 3 (Index-to-values map)**.** *The index-to-values map* $\mathcal{M} : Boolean^k \rightarrow (\mathcal{V} \rightarrow \mathbb{N})$ *assigns a bit vector of length* k *to a bag of values. The bag of values is itself a map that assigns each value to its number of occurrences.*

This representation summarizes the context-value pairs extracted from different projects by grouping together all values with a similar context.

B. Retrieval of Values for Test Generation

TestMiner provides values to a test generator. When the test generator retrieves values, e.g., to pass them to constructor or method call, it is crucial to a query in a timely manner because the time budget allocated for test generation is limited.

1) Integration into Test Generator: As a proof-of-concept, we integrate TestMiner into the state-of-the-art feedback-directed random test generator Randoop [37]. When Randoop requires a string value, e.g., to pass it as an argument to a method under test, we override its default behavior so that it queries TestMiner for V_{query} input values with a P_{query} probability.

2) Retrieval of Values: Algorithm 2 summarizes how TestMiner retrieves test input values for a given query with a context $\mathcal{C}_q$. At first, it assigns weights to the query context words in $\mathcal{C}_q$ using the following weighting function:

$$(0.5 + 0.5\frac{f_{t,q}}{max_t f_{t,q}}) \cdot log\Big(\frac{|D|}{|\{d \in D : t \in d\}| + 1}\Big) \quad (2)$$

where $f_{t,q}$ is the frequency of a context word in the query and all the other terms have the same meaning as in Equation 1. These weights give the same importance to the word frequency in the query and to its inverse document frequency in the corpus, effectively prioritizing uncommon words. The weighting function prevents bias towards longer signatures that may contain multiple similar words. TestMiner matches the query context $\mathcal{C}_q$ against contexts that have similar bit vectors using the search algorithm presented in [33]. The search function $searchSimhash$ returns a map $\mathcal{R} : String \rightarrow \mathbb{N}$ of indexed input values. This function selects values from hash indices that differ at most in $dist_{bits}$ bits from the Simhashinged query context. The threshold $dist_{bits}$ controls the number of input values returned to the test generator. A high value for $dist_{bits}$ matches against many contexts in the corpus, at the cost of high query latency and possibly unrelated values. Finally, the algorithm returns a map that represents a probability distribution across suggested values where the probability of a value is proportional to its frequency across all context-value pairs.

Algorithm 2 Retrieve values from index-to-values map.

Input: Query context $\mathcal{C}_q$ and index-to-values map $\mathcal{M}$
Output: Probability distribution $\mathcal{V}_{result}$ of values

1: $\mathcal{C}_{q,weighted} \leftarrow tfidfWeightsForQuery(\mathcal{C}_q)$
2: $\mathcal{R} = searchSimhash(\mathcal{C}_{q,weighted}, \mathcal{M}, dist_{bits})$
3: **return** $probDistribution(\mathcal{R}, \mathcal{M})$

III. Evaluation

A. Implementation

We integrate TestMiner into Randoop 3.0.8, a state-of-the-art feedback-directed, random test generator [37]. The retrieval part of our approach is implemented as a server application, making easy to integrate it into other test generators.
The static analyses part of TestMiner is implemented on top of the Eclipse JDT framework [1]. Integrating TestMiner into Randoop required only to change about 100 lines in the `ForwardGenerator` class and to add about 70 lines of new code to communicate with TestMiner.
To query TestMiner for values to be passed to a method m, the modified Randoop performs multiple queries with different contexts $\mathcal{C}_q$:

- the package, the class and the method name, and
- the class and the method name,
- the method name.

Querying with multiple contexts allows the test generator to retrieve more diverse values than with a single query because different queries emphasize different domain specific terms. We then combine all the returned values obtained from the three different queries into one set. Our implementation is publicly available at [3].

B. Experimental Setup

a) Data to Learn From: To learn about values suitable in a particular context, we apply TestMiner to 3,601 Java projects from the the Maven Central Repository [2]. We use all projects with source code of tests, which yields 263,276 string values used in 37,821 different contexts.
b) Classes Under Test: To evaluate the effectiveness of tests generated with TestMiner, we use 40 classes from 18 open source Java projects. 20 of these classes have been previously used for evaluating test generators [19], [35], [46], [47]. We further augment the existing benchmark classes with string manipulation classes from Defects4J [27] and with parsers of different text formats. Because the overall goal of TestMiner is to suggest values for classes beyond the corpus that the approach learns from, we remove from the corpus all the call site tuples that contain a type name in the projects of the classes under test.

c) Test Generation: For each class under test, we generate tests using both the default version of Randoop and the TestMiner-enhanced version. We use a time budget of 5 minutes per class and repeat each experiment 10 times to account for the random-based nature of Randoop, where each experiment uses a different random seed. Similar parameters were used in previous work [48]. We run the tests using JUnit and measure test coverage using the JaCoCo library. The coverage is the ratio of executed branches and all branches in the source code of a class. All experiments are performed on a machine with 48 Intel Xeon E5-2650 CPU cores and 64GB of memory.

C. Effectiveness of Generated Tests

To assess to what extent TestMiner increases the effectiveness of generated tests, we measure the branch coverage in each class under test. Figure 2 shows the average coverage over 10 test suites generated for each class. Overall, TestMiner improves the coverage for 30 classes and decreases it for 2 classes. On average over all classes, the relative improvement is 21%, increasing coverage from 57% to 78%. A Wilcoxon signed rank test shows this increase to be statistically significant ($p < 2.7 * 10^{-6}$). Cohen's d effect size, which is a statistical measure of the strength of the increase, is 0.87, which is considered large.

a) Increased Coverage: TestMiner increases coverage from 0% to more than 30% for the classes `PathURN`, `ResourceURL`, and `URN` because Randoop is unable to instantiate these classes. In contrast, by using domain-specific strings, TestMiner helps instantiate these classes, enabling the test generator to explore their behavior. However, for the `UTF8StreamJsonParser` class, TestMiner also fails to instantiate the class. Manual inspection shows that the class requires a complicated `IOContext` object that reads JSON files and assumes such files to exist. Providing such files is out of reach for both Randoop and TestMiner but may be addressed, e.g., by symbolic testing techniques [8].

b) Decreased Coverage: TestMiner decreases the branch coverage for 2 classes. A manual inspection of the produced test suites for `StringEscapeUtils` shows that the constant retrieval fails due to an implementation error in our prototype caused by a non-escaped unicode character that produces a communication error with the server. For `DateTime` and `StrTokenizer`, the decrease is on average, but achieves higher coverage than Randoop for some test suites, as shown by the error bars. Besides a few classes, most of the results do not show a significant difference in coverage across the 10 repetitions, i.e. the error bars are moderately small. The reason is the large timeout, which allows the test generator to reach saturation.

D. Query Result Examples

TestMiner provides to the test generator semantically rich and diverse values. The following values are examples from tests suites generated during our experiments:

1) *IBAN:* `SCBL0000001123456702`
2) *SQL:* `'abc' LIKE '_'`
3) *Network address:* `fe80::226:8ff:fefa:d1e3`
4) *E-mail:* `test@example.org`

However, there are also strings that do not help in testing a specific method, such as "foo" or "metadata". Fortunately, due to the nature of feedback-directed test generation, these values are likely to be ignored in later stages of the generation process. For example, Randoop filters already seen values that did not trigger any errors during execution.

E. Performance

a) Analysis and Indexing: The static analysis takes several hours to process the entire corpus but finishes within a single day. Indexing the context-value pairs takes about 20 seconds.

b) Retrieval: Retrieving values from TestMiner takes longer than using Randoop's hard-coded constants. To measure slows down of test generation, we compare the size of the test suites generated by Randoop and TestMiner. In the 5 minutes time budget, Randoop generates 545,895 tests, whereas the TestMiner-enhanced test generator creates only 243,494 tests, i.e., a 55% reduction. During our evaluation, millions of string values are requested by the test generator, but the number of unique queries is only 481, allowing our implementation to make extensive use of caching to keep the runtime overhead low. Overall, TestMiner slows down the test generation, but the increased runtime cost pays off because the tests generated with TestMiner are significantly more effective.

F. Influence of Parameters

TestMiner has three meta-parameters, which we set experimentally to maximize coverage increase:

- $dist_{bits} = 16$. Running TestMiner with $dist_{bits} = 4$, 8 significantly reduces branch coverage because fewer strings are returned to Randoop, which often defaults to its built-in strings. Setting $dist_{bits} = 32$ drastically increases query time and reduces the number of generated test in the time budget.
- $P_{query} = 0.5$. Running TestMiner with $P_{query} = 0.25,\ 0.5$, 0.75 provides a lower branch coverage.
- $V_{query} = 10$. Running TestMiner with $V_{query} = 5$, 15 provides no significant difference in branch coverage.

IV. Related Work

a) Test Generation: There are various approach for automatically generating test cases: symbolic [8], [28] and concolic [21], [45] execution [9], [52], [53], random-based test generation [16], [37], and search-based testing [17]. Beyond unit tests, automated testing has been applied, e.g., to concurrent software [13], [39] and to graphical user interfaces [12], [14], [23], [40]. Our work is orthogonal and could be integrated into many of these approaches.

b) Learning From Existing Code to Improve Test Generation: Liu et al. train a neural network to suggest textual inputs for mobile apps [32]. Similar to TestMiner, they learn from existing tests how to create test inputs. Our work differs by using information retrieval instead of a neural network, by learning

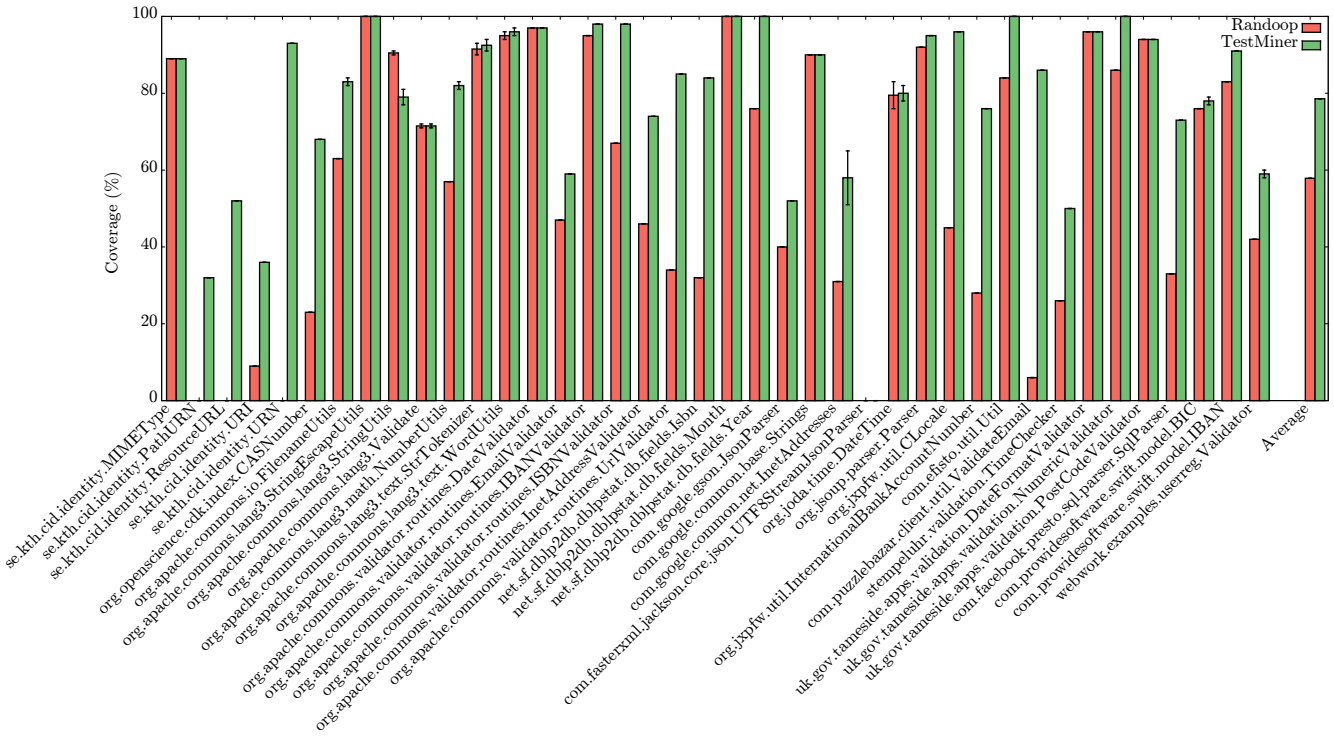

Fig. 2. Coverage without and with TestMiner. The colored bars represent the average coverage value over 10 runs of the test generator. The size of the error bars shows the difference between the first and the third quartile.

from already existing tests written by developers instead of writing tests specifically for learning, and by generating unit tests instead of UI-level tests.

Testilizer [36] mines UI tests for web applications by collecting input values for generating tests. In contrast, TestMiner statically collects input values from tests written for a different application. The "equivalence modulo input" (EMI) approach [30] tests compilers by modifying existing tests, i.e., programs, into new test. Our work also learns how to create new tests from existing tests but applies to various applications domains beyond compilers. Several approaches improve test generation by learning from existing code which methods to call [25], [50], [51]. TestMiner differs from these techniques by improving the selection of input values instead of the selection of calls.

c) Domain Knowledge for Test Generation: Studies show that providing domain knowledge to test generators improves testing effectiveness [18], [44]. Several approaches obtain domain-specific inputs, e.g., by querying web sites [35], [46], web services [7], or semantic knowledge graphs [34]. All these techniques require querying the internet for retrieving values. To the best of our knowledge, TestMiner is the first offline technique to suggest domain-specific input values.

d) Learning from Existing Source Code: Existing work exploits natural language information in source code, e.g., to detect programming errors [31], [38] and suboptimal identifier names [4], [24], to cluster software systems [15], to infer specifications [56], and to find inconsistencies between code and comments [49]. Our work also exploits domain knowledge encoded in natural language, specifically in identifier names, to improve testing. Other work on learning from existing code includes learning a statistical language model for code completion [43] and applying information retrieval to the problem of bug localization [26], [29], [41], [42], [57].

V. Conclusion

Test generation is a challenging problem, and finding suitable input values is an important part of this challenge. Our work presents TestMiner, a new approach that learns from a large corpus of existing tests which input values to use in newly generated tests based on the domain of the tested software. The approach combines static analysis and information retrieval to extract input values, to index them based on the context in which they occur, and to provide values suitable for a specific domain to a test generator. Our evaluation shows that TestMiner improves test coverage from 57% to 78%, on average over a set of 40 classes that challenge state-of-the-art test generators. The approach scales to thousands of analyzed projects, efficiently responds to queries for input values, and generalizes beyond the software analyzed as part of the corpus. The TestMiner approach provides a simple querying interface that enables existing test generators to benefit from domain-specific input values with little effort.

REFERENCES

[1] Eclipse JDT. http://www.eclipse.org/jdt.

[2] Maven Central. https://search.maven.org.

[3] TestMiner - Repository. https://github.com/lucadt/testminer.

[4] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Suggesting accurate method and class names. In *ESEC/FSE*, pages 38–49, 2015.

[5] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[6] M. Böhme and S. Paul. On the efficiency of automated testing. In *FSE*, pages 632–642, 2014.

[7] M. Bozkurt and M. Harman. Automatically generating realistic test input from web services. In *SOSE*, pages 13–24, 2011.

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

[9] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[10] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[11] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *PPOPP*, pages 43–52, 2014.

[12] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, 2013.

[13] A. Choudhary, S. Lu, and M. Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *ICSE*, 2017.

[14] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (E). In *ASE*, pages 429–440, 2015.

[15] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello. Investigating the use of lexical information for software system clustering. In *CSMR*, pages 35–44, 2011.

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

[17] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*, pages 416–419, 2011.

[18] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *ICST*, pages 121–130, 2012.

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

[20] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

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

[22] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[23] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *ISSTA*, pages 67–77, 2012.

[24] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP*, pages 294–317. Springer, 2009.

[25] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: object capture-based automated testing. In *ISSTA*, pages 159–170. ACM, 2010.

[26] H. Jiang, T. N. Nguyen, I.-X. Chen, H. Jaygarl, and C. K. Chang. Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management. In *ASE*, 2008.

[27] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, pages 437–440, 2014.

[28] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[29] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports. In *ASE*, 2015.

[30] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, pages 216–226, 2014.

[31] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *ICSE*, pages 1063–1073, 2016.

[32] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic text input generation for mobile testing. In *ICSE*, 2017.

[33] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.

[34] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Link: exploiting the web of data to generate test inputs. In *ISSTA*, pages 373–384, 2014.

[35] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *ICST*, pages 141–150, 2012.

[36] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging Existing Tests in Automated Test Generation for Web Applications. In *ASE*, pages 67–78, 2014.

[37] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE, 2007.

[38] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *ISSTA*, pages 232–242, 2011.

[39] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *PLDI*, pages 521–530, 2012.

[40] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*, pages 33–47, 2014.

[41] M. M. Rahman and C. K. Roy. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *ASE*, pages 220–225, 2016.

[42] S. Rao and A. Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *MSR*, pages 43–52, 2011.

[43] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, page 44, 2014.

[44] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Softw Test Verif Reliab*, 26(5):366–401, 2016.

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

[46] M. Shahbaz, P. McMinn, and M. Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *QSIC*, pages 79–88, 2012.

[47] A. Shahbazi and J. Miller. Black-box string test case generation through a multi-objective optimization. 42(4):361–378, 2016.

[48] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T). In *ASE*, pages 201–211, 2015.

[49] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments?*/. In *SOSP*, pages 145–158, 2007.

[50] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *TAP*, pages 77–93, 2010.

[51] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *ESEC/FSE*, pages 193–202, 2009.

[52] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA*, pages 189–206, 2011.

[53] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.

[54] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *PLDI*, pages 283–294, 2011.

[55] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *ESEC/FSE*, pages 114–124, 2013.

[56] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.

[57] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang. Learning to Rank for Question-Oriented Software Text Retrieval. *ASE*.