Combinatorial Generation of Structurally Complex Test Inputs for Commercial Software Applications

Hua Zhong[1,2], Lingming Zhang[3], Sarfraz Khurshid[2]
[1]Google Inc, 94043, USA, zhongh@google.com
[2]The University of Texas at Austin, 78712, USA, {hzhong,khurshid}@utexas.edu
[3]The University of Texas at Dallas, 75080, USA, lingming.zhang@utdallas.edu

ABSTRACT

Despite recent progress in automated test generation research, significant challenges remain for applying these techniques on large-scale software systems. Large-scale software systems under test often require structurally complex test inputs within a large input domain. It is challenging to automatically generate a reasonable number of tests that are both legal and behaviorally-diverse to exercise these systems. Constraint-based test generation is an effective approach for generating structurally complex inputs for systematic testing. While this approach can typically generate large numbers of tests, it has limited scalability – tests generated by this approach are usually only up to a small bound on input size. Combinatorial test generation, e.g., pair-wise testing, is a more scalable approach but is challenging to apply on commercial software systems that require complex input structures that cannot be formed by using arbitrary combinations. This paper introduces comKorat, which unifies constraint-based generation of structurally complex tests with combinatorial test generation methods. Specifically, comKorat integrates Korat and ACTS test generators to generate test suites for large-scale software systems with structurally complex test inputs. We have successfully applied comKorat on four large-scale software applications developed at eBay and Yahoo!. The experimental results show that comKorat outperforms existing solutions in execution time and test coverage. Furthermore, comKorat found a total of 59 previously unknown bugs in the above four applications.

CCS Concepts

•**Software and its engineering** → Software testing and debugging;

Keywords

Constraint-based test generation, Combinatorial test generation, Korat, ACTS

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2983959

1. INTRODUCTION

Automated test generation techniques have made much progress in the last decade [1, 2, 10, 13]. Various techniques for symbolic execution [5, 7], bounded-exhaustive testing [4, 14], combinatorial testing [11, 12], and random testing [6, 8, 9, 16], form a part of the state of art [1]. Typical white-box test generation techniques rely on various source code analysis and are largely impacted by the scale of the program under test. To illustrate, symbolic execution, which enables white-box test generation by exploring program paths using symbolic inputs, performs a powerful analysis but is limited by the complexity of the operations of the program under test and is commonly applied for testing units of code with moderate size. In contrast, black-box testing techniques like bounded-exhaustive testing [4, 14], combinatorial testing [11, 12] and random testing [6, 8] focus only on the test data specifications or testing requirements. They generate tests without considering the actual implementation of the program under test, and are thus less sensitive to the complexity of the program under test.

While combinatorial and random approaches are more scalable (in terms of the size and complexity of code that they can apply to) and easy to use, these techniques cannot be directly applied to create diverse suites of structurally complex test inputs. An example that arises when testing commercial applications like the ones that we considered is the integration testing of web service applications. The input to the program usually has complex structure (e.g. JSON) and cannot simply be sampled at random. Testers often write these inputs in text files manually and then the test execution framework parses the files and converts them to input objects to execute the tests.

Bounded exhaustive testing is then intensively studied to generate structurally complex test inputs. More specifically, constraint-based testing, a representative bounded exhaustive testing methodology, has been demonstrated to be capable of finding bugs in various real-world software applications, including resource discovery architecture, XPath compiler, and so on. To illustrate, Korat [4, 14] is a Java-based approach for constraint-based testing of structurally complex test inputs. Korat performs specification-based testing – given a Java predicate that describes properties of desired input data, Korat performs a backtracking search to explore the input space of the predicate and enumerates all inputs for which the predicate returns true. Korat returns each enumerated input as a desired test input. To test a program, Korat requires the program precondition to generate tests and the postcondition to verify correctness of the program.

A key strength of constraint-based test generation, is that it can systematically generate structurally complex inputs [4, 14]. While this approach is very effective on some programs, scaling it to commercial software applications is challenging for two reasons: 1) the underlying search spaces are usually too large to be explored exhaustively; 2) systematic generation will likely create an enormous number of tests, which are impractical to run.

A key strength of combinatorial testing [11, 12] is its effectiveness at reducing test input combinations. It usually enables a significant reduction in the number of test cases without compromising much functional coverage. For example, pair-wise testing is often treated as a reasonable cost-benefit compromise between likely computationally infeasible higher-order combinatorial testing methods, and less exhaustive methods which fail to exercise all possible pairs of parameters.

In this paper, we present comKorat [17, 18], an approach that combines ideas from constraint-based test generation and combinatorial testing and applies them in synergy to benefit from their strengths. Specifically, comKorat builds on top of the Korat [4, 14] test generator, which generates non-equivalent input structures using imperative constraints, and the ACTS [11, 12] combinatorial testing tool, which generates combinatorial tests to populate the structures generated by Korat. We evaluate comKorat by using it to generate tests for 4 applications developed at eBay and Yahoo!. The experiments show that it is feasible to use comKorat to generate tests for commercial software. Furthermore, the test suites generated by comKorat outperform the traditional Korat approach as well as existing manual test suites written by test engineers at the two companies.

This paper extends our recent tool paper [18] in three directions. First, it formally presents the comKorat approach and its core test generation algorithm. Second, it evaluates comKorat on three new large-scale industrial applications developed at eBay. Third, it evaluates comKorat for two new types of testing (UI testing and API testing). This paper also presents more details of the core approach. We make the following contributions: (1) We introduce the idea of integrating constraint-based test generation with combinatorial testing; (2) we embody our idea in comKorat by integrating Korat with ACTS; (3) we conduct an evaluation based on commercial software applications developed at eBay and Yahoo!. The experimental results show that comKorat can substantially reduce the search space and the number of structurally complex tests generated as well as providing high code coverage. The test suites generated by comKorat detected 59 previously unknown defects.

2. MOTIVATING EXAMPLE

In this section, we use a slightly simplified example from one of the benchmark systems to illustrate the comKorat approach for generating structurally complex inputs. This example demands the test generation algorithm to explore a generic tree data structure (master tree) and return a set of subtrees from the master tree. These subtrees serve as test inputs to the system.

As shown in Figure 1, a master tree has 4 types of nodes: a Root, a set of L nodes, a set of D nodes, and a set of B nodes. The master tree has a few constraints: (1) B nodes can only be leaf nodes in the master tree; (2) the parent of a B node is an L node; (3) the parent of an L node is either the Root or a D node; (4) the parent of a D node is an L node; (5) the children of Root are L nodes; (6) in a given level of the master tree, all nodes are of the same type.

A valid test case is a subtree of the master tree and there are also some constraints on the test cases: (1) All test case trees have the same root as the master tree; (2) A test case tree must contain two or more B nodes; (3) A test case cannot contain two B nodes from the same parent; (4) All leaf nodes in a test case tree are B nodes; (5) If two B nodes have two different D nodes as their ancestors and those two D nodes share the same parent, these two B nodes are in conflict with each other and thus cannot be in the same test case; (6) A parameter *size* is introduced to limit the number of B nodes (leaves) in a test case. When *size* is equal to 3, it means all generated subtrees have up to 3 B nodes.

The ideal test suite should cover all possible combinations of B nodes. A naive implementation of such a test suite could exhaustively enumerates all possible subtrees in the master tree to cover the above combinations. However, most of the exhaustively generated subtrees will violate some of the structural constraints described above and will become invalid inputs to the system.

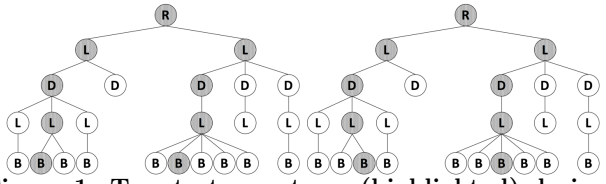

Figure 1: Two test case trees (highlighted) derived from the same master tree. Note that both trees have same structures except on the leaf level

Although Korat offers a possible solution to remove illegal inputs, it faces several challenges to be effective in practice. The first challenge is that the execution time of Korat can be up to days to generate valid inputs. In the real world application there are usually hundreds of B nodes in the master tree and Korat needs to explore billions of possible structures to find all legal candidates. The second challenge is that the valid input domain could contain millions of possibilities and it's infeasible to execute such a large test suite. Our comKorat approach is introduced to resolve the above issues by separating the input structure generation and input value generation to reduce the search space for Korat. comKorat also adopts combinatorial test selection approach to reduce the total number of valid tests.

We notice that most of the nodes of a master tree are B nodes; thus the subtrees generated by Korat can be categorized into a small number of groups based on their structure similarity and subtrees inside one group have almost identical structures except in the leaf layer (Figure 1). If we consider the leaf nodes as values of their parent nodes and remove them from the master tree, then the input space reduces drastically and Korat needs to explore only a few thousand subtree candidates instead of millions. We can then adopt combinatorial method to fill in those values (leaf nodes) to reduce generation time and test suite size.

3. APPROACH

The overall comKorat algorithm is shown in Algorithm 1. The inputs to the algorithm are the program under test

ALGORITHM 1: Main Algorithm

Input: P – the program under test
Input: I – the input class
Input: TW – the degree of interactions
Output: F – the final test suite file

```
1  TestSuite TS ← ∅
2  InputStructureClass S = removeInvariantFields(I)
3  HashMap MAP = genParamMap(I)
4  GeneratedStructure G = genKoratTests(S, MAP)
5  for each GeneratedStructure BS in Korat generated
   structures G do
6  |  List L = retrieveParams(BS)
7  |  Test TestSet = genCombinatoryTest(L, TW, MAP)
8  |  for each test T in combinatorial test set TestSet do
9  |  |  TestCase TC = AddValue2Struc(T, BS)
10 |  |  TS.add(TC)
11 |  end
12 end
13 TestInputFile F = convert(TS)
```

P, an input class I that contains the structure and input domain of input data, and finally a degree of interaction parameter TW for the combinatorial test generation. The algorithm first initializes an empty TestSuite set TS at line 1. comKorat then identifies the structural variants in the input class and removes all other fields from the class to create an input structure class S at line 2 (e.g., removing B nodes from master tree for our motivating example shown in Section 2). The input parameters and their values are preserved in a HashMap MAP. This new input class S is then passed to Korat to generate input structures G. In the next step, for each generated structure BS, a list L is created to store the values of parameters inside BS. ACTS is applied to generate a combinatorial test set $TestSet$ using value list L, variable map MAP and the interaction strength parameter TW at line 7. For each test T in test set $TestSet$, the variables inside T are used to update structure BS to generate a complete test case TC at line 9. This test case is then added to the test suite TS at line 10. Finally, the test suite (Java instances) are converted to a desired format and stored to local file system at line 13.

3.1 Structure/Value Separation

The beginning stage of comKorat is to separate the input values from the input structures. In our approach, we identify a set of variables from an input class as structural invariants. A variable inside a class is a structural invariant when it's value has no influence on the structure of the class. We remove these variables and create a new input class for structure generation.

We use HashMaps to store the removed variables. For each removed variable, we use it's parent node in the structure as the key of the map, and make the variable as the value. Variables sharing the same parent node will be put into a list, and the orders of the variables are preserved in the list. For JSON and XML inputs, we map the names of invariant variables to their values in the HashMap. We utilize boundary value analysis [15] to select representing values for those variables with large input domains (e.g. select value 1 and 1000 for an Integer field with input range from 1-1000)[1]. We will illustrate this approach later in Section 3.3.

```
public boolean repOK() {
    if (array==null||size < 0
        ||size != array.length)
            return false;
    // check if base layer is included
    if (BASE_LAYER==1 && !repOK5())
            return false;
    // enforce ordering
    if(repOK4())
            return false;
    //check duplicated elements
    if(repOK3())
            return false;
    //check conflicting elements
    if(repOK2())
            return false;
    return true;
}
```

Figure 2: repOk example

3.2 Input Structure Generation

With a new input class, Korat can then search on a much reduced input space. The next step is to implement Korat's repOK and finitization methods for structure generation. The following subsections only elaborate on a few characteristics of implementing the two methods.

repOK Inputs to commercial software tend to have complex structures and large input domains. Even after taking a structure/value separation step, the search space for the new input class can still be very large. To optimize this search process, we follow prior work [3] to arrange our checks in the repOk method in such an order that a previous check can always prune a larger portion of the search space than a later check. In Figure 2, we show an example of the repOK method implemented for Yahoo product A. In this input tree structure, each leaf node (L node) is represented by an unique id (an integer value to allow ordering). As we can see in the Figure 2, before checking conflicting elements, the repOk method checks if the base layer is included, keeps only ordered candidates, and then eliminates candidates which contain the same set of L nodes but in different order. Those checks remove a significant part of the search space and leave only a few candidates for the repOK2 method to check for nodes conflicts. This conflict check is performed as the last check in repOK because it is very time consuming. This conflict check needs to find the paths from the leaf L nodes to the Root of the master tree and check all nodes in the path to see if the two L nodes violate the conflict constraint. Arranging the checks in this manner could largely increase the performance of candidate search.

Finitization The next task comKorat needs to do is to bound the variables (non-invariants in new input class) in finitization method. As mentioned above, comKorat only uses Korat to generate structures. Therefore, the finitization methods only needs to bound those non-invariant variables. One challenge we encountered is that the input range of these variables are usually quite large (e.g. an integer field with input range from 0-1000). If we use finitization method to bound these variables directly, Korat could end up producing a large number of structures. To resolve this

[1]Note that for the fair comparison, we adopt the same settings for both Korat and comKorat.

issue, we utilize boundary value analysis [15] to select representing values for each variable that each of the selection produce exactly one unique structure. While this step might sound complicated, it is actually quite straight forward since these key values are already defined in the repOK methods to check for the structural constraints. To further explain our approach, let's walk through a concrete example in the rest of this section. In eBay product B, there is an integer variable and we name it as ITP. The input domain of this variable could range from 0 to an arbitrary large number. If the ITP is greater or equal to 500, then another field in the input class which has a tree structure must have 2 nodes on the second level of the tree. Since only a selective number of values can alter the structure of the input class, a set of boundary values are chosen from the repOK method to replace the continuous input range. In this example, we end up choosing five values of (0, 1, 499, 500, 1000) to represent the input range of the variable ITP. Note that the setting is the same for both Korat and comKorat.

With the implemented repOK and finitization methods, Korat is then adopted to generate input structures and each generated structure is passed to the combinatorial test generator to create a number of t-way combinatorial tests.

3.3 Combinatorial Tests Generation

Given a generated base structure (e.g., Java instance), ACTS is brought in to generate the final tests for the programs. Figure 3 illustrates the main steps of the combinatorial generation phase. comKorat first gathers the invariant variables (Java String) and finds them in the HashMap. Alongside a set of constraints, these variables and their values are passed to ACTS for generating t-way combinatorial tests. Finally, each combination of values is added back to a copy of the base structure to create a complete input instance. Rest of this section elaborates on value selection, value constraints and input instance creation.

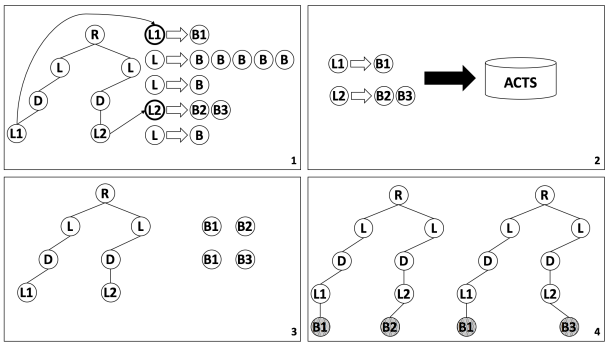

Figure 3: Illustration for combinatorial test generation: 1) mapping the base structure to a subset of variables in HashMap; 2) extracting the list of variables from HashMap and passing to ACTS; 3) ACTS generated combinations; 4) complete test case generated

Since only a selective number of values can alter the execution path of the programs, a set of values are selected for each variable with a large input domain using system requirements and all those values are stored in the HashMap. However, applying the combinatorial test method to those variables directly could still produce a test suite with millions of test cases. These test suites are still too large to fit into current test execution framework. Besides, many of the tests are considered as negative tests and have very low priorities. Executing such a large number of negative tests is ineffective and time consuming since a lot of them exercise the same part of a system.

To further reduce the number of negative test cases generated by ACTS, we add a series of constraints in ACTS to remove them from the final results. For example, a combination of currency code and country code must be in sync to produce a valid input. We added such a constraint in our ACTS implementation to make sure the invalid combinations are excluded from the final results. We also setup a time threshold for each test execution framework and the framework will timeout if it can't execute all the generated tests within a given number of hours. We keep tuning the interaction strength parameter t in ACTS until a proper number of tests are generated for the benchmark systems.

4. EXPERIMENTAL EVALUATION

4.1 Programs Under Test

We applied comKorat on four applications: three of them developed at eBay and one at Yahoo!.

eBay product A. eBay product A is an UI application and provides our subject for UI testing. The test suite is stored in text files. Automated tests parse the text files and convert them to Java instances to execute the test.

eBay product B. eBay product B is an API program and provides our subject for API testing. The test cases are in XML. The automated tests are typical API functional tests.

eBay product C. eBay product C is a Java application and provides our first subject for backend/integration testing. The test suite is stored as Java Objects and it verifies system integrations between different components of the product.

Yahoo! product A. Yahoo! product A is a backend system developed for Yahoo! search engine and provides our second subject for backend/integration testing. The system takes a tree structure as its test input. Our tool paper [18] presents the results for this product, which we include here for completeness.

4.2 Experimental Setup

The experiments discussed in this paper are performed on a Mac machine with a 2.5 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM running JVM 1.8.0-45 on OS X Yosemite 10.10.1.

4.3 Result Analysis

Table 1 presents the results of test generation by comKorat. Empirical investigations suggest 90% of the problems can be triggered by the interaction of three or fewer parameters in commercial web applications [11,12]. We also found out that comKorat will either generate an enormous large number of tests or timeout during test generation when exercises high degree interaction combinations (4-way or above). Therefore, we tried 2-way and 3-way combinatorial generation for the applications at eBay and 2 to 4 way combinatorial generation for the application at Yahoo!.

In the table, Column 1 denotes the studied benchmarks; Column 2 lists the size parameter specific to the Yahoo! product; Column 3 presents the number of structures generated by comKorat while Column 4 presents the number of candidate structures explored during the generation; finally,

Table 1: comKorat's performance on several benchmarks. *size* is a parameter only in Yahoo! product A.

benchmark	size	struc. gen.	candi. expl.	combinatory tests		
				2-way	3-way	4-way
eBay product A	n/a	39	124	568	8150	n/a
eBay product B	n/a	42	108	624	5831	n/a
eBay product C	n/a	14	178	356	3418	n/a
Yahoo! product A	3	31	71	16970	32114	32114
	4	458	2521	260214	500312	826404
	5	4298	66221	2210309	4883621	10383222

Table 2: Performance comparison. For each benchmark, performances of comKorat (use 2-way combination) and Korat are compared.

		comKorat		Korat	
benchmark	size	test gen. (2-way)	total time	test gen.	total time
eBay product A	n/a	568	0.246s	386695	11.972s
eBay product B	n/a	624	0.303s	387175	17.146s
eBay product C	n/a	356	0.243s	152615	7.893s
Yahoo! product A	3	16970	2.533s	32114	3.431s
	4	260214	4.285s	826404	17.72s
	5	2210309	10.946s	27653142	986.63s

Columns 5 to 7 present the number of final tests generated based on 2-way to 4-way combinatorial value generation. comKorat can generate all tests even for very large state spaces because the separation of structure generation and value generation allows Korat to explore only a tiny fraction of the input space. Shown in Column 4 of Table 1, comKorat only needs to check a small number of candidates during it's structure generation stage. The number of structures generated by Korat is also very small (Column 3), and ACTS is brought in to search rest of the space to reduce the number of generated tests. Without ACTS, Korat would generate infeasibly many tests.

Table 2 compares the performance between comKorat and Korat. For all benchmarks, we compare the total number of tests and the time to generate them for a range of parameter values. From Table 2, we can see that comKorat significantly reduced the number of tests as well as test execution time. To illustrate, for Yahoo! product A, comKorat largely reduced the generated test number from 27,653,142 to 2,210,309, and allowed the system to explore up to 4-way combinations in less than 11 seconds. During our experiment, we also found out that it is not feasible to execute Korat generated tests directly as these large test suites will cause the execution framework to timeout. However, the performance of Korat test generation is quite robust, e.g., Korat generated 387175 tests in less than 20 seconds.

Table 3 compares the performance between comKorat generated suites with existing manually generated test suites. Note that we were unable to collect the exact manual test construction time due to the complexity of the manual testing process; also note that for Yahoo! product A, we were unable to obtain the code coverage information due to the difficulty to instrument the code in the integrated environment. From the table, comKorat generated suites outperform legacy suites in code coverage. Furthermore, comKorat also reveals new defects in the studied programs. After the comparison, we can find that automated test generation using comKorat not only removes the laborious human effort, but also reduces human bias and thus can achieve higher code coverage and fault detection rate.

4.4 Defect Study

To further understand the performance of comKorat, we manually select four defects to analyze why comKorat outperforms the existing solutions and why it is infeasible to use only Korat or ACTS to generate those tests.

Defect 1. In eBay product A, three tests failed because of a malfunction in a button-click event. The root cause of the failure is that a combination of two parameters in the input instance enables a security flag on an object and the button is used to update information on the object. The underlying system uses two different requests to retrieve a list of objects for displaying and for updating. While the flagged object is present in the response of first request, it is not in the second one. Thus end users can still see the object in UI but once the user clicks on the Update button, the button event crashes and freezes the UI. Since the input instance has complex structures and only a few combinations of the two parameters can trigger the security flag (3 out of 568), it is difficult for a human to create these inputs.

Defect 2. In eBay product B, backend system gives wrong level to a list of objects in it's response. The root cause is that the program failed to calculate the value of a variable $p1$ for some currency and location combinations. System decides the level of the objects based on the value of $p1$ and the locations of the object. One combination generated by comKorat uncovered this defect. Existing tests only cover a few combinations and failed to uncover this defect.

Defect 3. In eBay product C, system returns a notification message which is designated for US on the UK and Germany sites. The root cause of this defect is that the system failed to add condition check when returning this notification message on UK and Germany sites. In a correct behavior, different messages should not be shown on the pages. Similar to the above case, comKorat created inputs which invoke this message on UK and Germany sites. Since the condition of showing such a message is very complicated, existing tests don't cover this message on all sites.

Defect 4. In Yahoo! product A, 3 test case trees caused backend system to return empty responses for the given inputs. These B nodes combinations ($size$=3) caused backend system unable to locate information in data store to rank a particular search result and the system times out searching for the information. As we introduced in above sections, existing solution can not generate t-way combinations on test case trees and failed to uncover the issue.

In summary, comKorat is able to generate t-way combination tests for structurally complex test inputs. comKorat

Table 3: Performance comparison between existing manual test suites and comKorat.

	comKorat				Existing solution		
benchmark	# tests	coverage	# new defects	time	# tests	coverage	time
eBay prod. A	568	95%	5	0.246s	315	85%	n/a
eBay prod. B	624	100%	3	0.303s	277	91%	n/a
eBay prod. C	356	83%	28	0.243s	128	78%	n/a
Yahoo! prod. A	260214 ($size$=4)	n/a	23	4.285s	5815 ($size$=2)	n/a	n/a

outperforms existing solutions in code coverage and fault detection. Given a small number of t ($t \leq 4$), comKorat can generate effective test suites for commercial software while significantly reducing the number of tests. However, it is still hard to use comKorat to enable extremely thorough testing of applications with manageable numbers of tests, and we plan to further explore this direction in our future work.

Threats to Validity. *External validity.* Although we used 4 programs, the differences seen in our study may be difficult to generalize to other programs. Further reduction of these threats requires additional studies involving additional object programs. *Internal validity.* There may be faults in our implementation of the Korat structural generation and ACTS test generation, as well as the other controlled techniques. To reduce this threat, we reviewed all the code that we produced for our experiments and we test our code with some small examples to verify the correctness of the test generation before conducting the experiments. *Construct validity.* The effectiveness results may greatly differ if different programs are used in the experiments. Depending on the length of service, programs that have been served for a longer time tend to have less undetected defects, as most of the defects have been exposed during it's lifespan. One of the programs (eBay product C) was a new product and thus our test suites discovered a lot of issues from the program.

5. CONCLUSIONS

This paper presents comKorat, a test generation technique for structurally complex test inputs. The key idea of comKorat is to integrate the strengths of two approaches for test generation: constraint-based generation and combinatorial generation, which are traditionally employed separately. Specifically, comKorat builds on Korat and ACTS to embody a synergistic approach. We applied comKorat to test 4 commercial applications developed at eBay and Yahoo!. The experimental results show that it is feasible to automate generation of test cases for such applications, even when the search space for inputs is very large. The experiments also show that comKorat achieved higher code coverage than the existing solutions adopted by the host companies. Furthermore, comKorat detected 59 previously unknown bugs in these applications.

6. ACKNOWLEDGMENTS

This work is partially supported by the NSF Grant Nos. CCF-1319688 and CCF-1566589. Lingming Zhang also gratefully acknowledges his Google Faculty Research Award.

7. REFERENCES

[1] S. Anand, E. Burke, T. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, and P. Mcminn. An Orchestrated Survey on Automated Software Test Case Generation. *JSS*, 2013.

[2] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-Oriented Programs. In *M-TOOS*, 2006.

[3] V. Bengolea, N. Aguirre, D. Marinov, and M. F. Frias. Using Coverage Criteria on RepOK to Reduce Bounded-exhaustive Test Suites. In *TAP*, 2012.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *ISSTA*, pages 123–133, 2002.

[5] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice—Preliminary Assessment. In *ICSE*, 2011.

[6] T. Y. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Advances in Computer Science*. 2004.

[7] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Test Generation for Graphical User Interfaces Based on Symbolic Execution. In *AST*, 2008.

[8] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and R. Kuhn. An Empirical Comparison of Combinatorial and Random Testing. In *ICST*, 2014.

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

[10] N. Havrikov, M. Höschele, J. P. Galeotti, and A. Zeller. XMLMate: Evolutionary XML Test Generation. In *FSE*, 2014.

[11] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial Software Testing. *IEEE Computer*, 42(8):94–96, 2009.

[12] R. Kuhn, Y. Lei, and R. Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10(3):19–23, 2008.

[13] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*, 2001.

[14] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel Test Generation and Execution with Korat. In *FSE*, 2007.

[15] T. Murnane, K. Reed, and R. Hall. On the learnability of two representations of equivalence partitioning and boundary value analysis. In *ASWEC 2007*, 2007.

[16] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback Directed Random Test Generation. In *ICSE*, 2007.

[17] H. Zhong. Pairwise-Korat: Automated Testing Using Korat in an Industrial Setting. *Master's Report, The University of Texas at Austin, USA*, 2015.

[18] H. Zhong, L. Zhang, and S. Khurshid. The comKorat Tool: Unified Combinatorial and Constraint-based Generation of Structurally Complex Tests. In *NFM*, 2016.