Reducing Combinatorial Testing Requirements Based on Equivalences with Respect to the Code Under Test

SARFRAZ KHURSHID, The University of Texas at Austin
DARKO MARINOV, University of Illinois at Urbana-Champaign

Combinatorial testing, where different combinations of parameter values are used to create test inputs, is a well-known approach for black-box testing of software systems. Researchers have defined several coverage criteria for combinatorial testing. Among them, the most comprehensive criterion is all combinations coverage. While using this criterion gives the most assurance in the correctness of the code under test, this criterion can have too many test requirements, which can make it impractical to apply. This paper introduces a new, simple approach that provides the same assurance as all combinations coverage but typically with fewer test inputs, thereby reducing the overall cost of combinatorial testing. Our key insight is that one test input execution can cover several test requirements for combinatorial coverage criteria. Our approach builds on the Korat test-generation technique to explore which combinations of parameter values are equivalent with respect to the code under test. An illustration on a pedagogical example shows how this approach can lead to substantial reduction in the number of tests.

1. INTRODUCTION

Combinatorial testing (also known as "N-wise testing" or "combinatorial interaction testing") is a well-studied approach for black-box testing of software systems [Grindal et al. 2005; Ammann and Offutt 2008; Nie and Leung 2011]. For example, two surveys of combinatorial testing [Grindal et al. 2005; Nie and Leung 2011] cover dozens of techniques and are themselves cited over 430 times each (according to Google Scholar on July 15, 2018). Combinatorial testing requires the user to provide domains of values for each parameter input of the software system under test, and then creates test inputs by different combinations of parameter values.

Researchers have defined several coverage criteria for combinatorial testing. The criteria range from the weakest base-choice, to stronger and widely used pairwise criterion (also known as "all-pairs criterion"), to even stronger N-wise (for $N > 2$), up to the most comprehensive criterion all combinations coverage. As the criterion name implies, all combinations coverage requires the entire cross-product of all parameter values from the input space to be used for testing. While using this criterion gives the most assurance in the correctness of the code under test, this criterion can have too many test requirements, which can make it impractical to apply. For example, if there are p parameters, each with only 2 values, then the total number of combinations is 2^p.

This paper introduces a novel, simple approach that provides the same assurance as all combinations coverage but typically with fewer test inputs, thereby reducing the overall cost of combinatorial testing.

This research was partially supported by the US National Science Foundation Grant Nos. CCF-1409423, CCF-1421503, CCF-1704790, and CCF-1718903.
Authors' addresses: Sarfraz Khurshid, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA; Darko Marinov, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. Email correspondence: {khurshid@utexas.edu,marinov@illinois.edu}

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

Our key insight is that execution of one test input can cover several test requirements for combinatorial coverage criteria. In other words, multiple test requirements that are non-equivalent at the black-box level (which does not consider the code under test but only its specification) may be equivalent at the white-box level (which does consider the code under test). The key challenge is to determine which combinations of given input parameter values have equivalent behavior for the given code under test.

To address this challenge, the approach presented in this paper builds on the Korat test generation [Boyapati et al. 2002]. Korat is a technique for generating structurally complex test inputs. Given a *predicate* (i.e., a function that takes one input and returns a Boolean value) and a *finitization* (i.e., a specification that bounds the size of some input components or provides concrete values for other input components), Korat can generate all (non-isomorphic) inputs that satisfy the given predicate (i.e., for which the predicate returns "true") within the input space defined by the finitizations. For example, Korat can be used to generate all red-black trees up to a given maximum size by providing a predicate that checks whether an input graph is a red-black tree and a finitization that bounds the number of nodes and provides values for the node color (red or black). Korat has been used to test various kinds of software (e.g., detecting 59 previously unknown bugs in products of eBay and Yahoo! [Zhong et al. 2016a; 2016b]), but was not applied to reduce the number of tests created by combinatorial testing.

In this paper, we demonstrate how Korat can be applied to explore which combinations of parameter values in combinatorial testing are equivalent with respect to the code under test. Before we summarize our new approach, we first briefly describe how Korat operates. Korat searches through the (finite) input space defined by the finitization. Korat instruments the predicate to monitor accesses that the predicate makes to parts of the input. Korat then runs the predicate for some inputs and uses the information from monitoring to prune large parts of the search space. As a result, Korat can effectively explore very large state spaces, generating all inputs that satisfy the predicate but without explicitly enumerating all elements of the input space.

To apply Korat to combinatorial testing of a piece of code, namely a function with multiple parameters, requires (1) turning the function into a *predicate* that conceptually takes one complex input (e.g., one object whose fields represent the parameters of the original function), invokes the function by passing the actual function arguments (from the input object's field values), and returns "true"; and (2) using the domains for parameter values provided for the original function as the *finitization* for Korat. With such changes, we can run Korat on the newly constructed predicate and map the search information from Korat back to the original combinations to determine which combinations result in the same output of the original function.

We illustrate the newly proposed approach on a pedagogical example of the triangle classification function [Ammann and Offutt 2008]. This function is widely used to introduce basic concepts of software testing. The function takes as input three integers that are supposed to represent the lengths of the triangle sides, and returns as output the type of triangle (e.g., "equilateral" if all sides are the same non-negative number, or "invalid" if the lengths do not represent a valid triangle). A widely used textbook on introduction to software testing [Ammann and Offutt 2008] shows how combinatorial testing can be applied on this function by using values -1, 0, 1, and 2 for the parameters and exploring various combinations, up to all $4^3 = 64$ combinations.

We show how using our new approach can lead to a substantial reduction in the number of combinations *without* losing any exhaustiveness and guarantees provided by all combinations. In other words, our approach determines which combinations are *definitely equivalent* for the given code and does not just randomly or systematically sample some subset of the combinations. In this particular example, our approach finds that only 22 combinations are non-equivalent (while the remaining 42 are equivalent). We also compare our approach, based on concrete exploration done by Korat, with the approach based on symbolic execution, a traditional approach to test generation [King 1976; Clarke 1976] that

is subject of many active research investigations. While symbolic execution, as expected, finds even more equivalent paths than our Korat-based approach, note that symbolic execution comes at a great runtime cost, unlike our Korat-based approach.

The main contribution of this paper is to demonstrate how to reduce the number of combinations for all combinations coverage using the approach of Korat's pruning. We believe our work lays the foundation for a new approach to reducing the cost of combinatorial testing, not just for all combinations coverage, but also for other weaker but widely used criteria, such as pair-wise coverage.

2. OVERVIEW

This section uses an illustrative example to give an overview of our approach and compare its results with two previous approaches: traditional combinatorial testing with all combinations coverage, which is a black-box approach, and symbolic execution [King 1976; Clarke 1976], which is a white-box approach. We use the well-known triangle classification problem as our illustrative example, which has been widely studied in the software testing literature. Specifically, we use the Java implementation written by Jeff Offutt [Ammann and Offutt 2008], with the following method being the main triangle classification routine:

```
private static int Triang (int Side1, int Side2, int Side3)
{
    int tri_out;
    // tri_out is output from the routine:
    //    Triang = 1 if triangle is scalene
    //    Triang = 2 if triangle is isosceles
    //    Triang = 3 if triangle is equilateral
    //    Triang = 4 if not a triangle

    // After a quick confirmation that it's a legal
    // triangle, detect any sides of equal length
    if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
    {
        tri_out = 4;
        return (tri_out);
    }

    tri_out = 0;
    if (Side1 == Side2)
        tri_out = tri_out + 1;
    if (Side1 == Side3)
        tri_out = tri_out + 2;
    if (Side2 == Side3)
        tri_out = tri_out + 3;
    if (tri_out == 0)
    {  // Confirm it's a legal triangle before declaring
        // it to be scalene

        if (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
                Side1+Side3 <= Side2)
            tri_out = 4;
        else
            tri_out = 1;
        return (tri_out);
    }
```

```
        /* Confirm it's a legal triangle before declaring  */
        /* it to be isosceles or equilateral  */

        if (tri_out > 3)
            tri_out = 3;
        else if (tri_out == 1 && Side1+Side2 > Side3)
            tri_out = 2;
        else if (tri_out == 2 && Side1+Side3 > Side2)
            tri_out = 2;
        else if (tri_out == 3 && Side2+Side3 > Side1)
            tri_out = 2;
        else
            tri_out = 4;
        return (tri_out);
    } // end Triang
```

The method `Triang` takes three integer inputs—`Side1`, `Side2`, and `Side3`—and returns an integer result. Each input represents a length, and the method determines whether the three lengths can form a valid triangle and if so, what kind of triangle it is. Specifically the method returns:

—1 if the triangle is scalene, i.e., all 3 sides have different lengths;

—2 if the triangle is isosceles, i.e., exactly 2 sides of have the same length;

—3 if the triangle is equilateral, i.e., all 3 sides have the same length; and

—4 if the three input lengths cannot be the lengths of sides of a valid triangle.

2.1 Traditional all combinations coverage

We first apply the traditional all combinations coverage criteria to define the test requirements for `Triang`. Following the standard black-box testing methodology, for each input parameter, we first partition the domain of all possible values for the parameter into a finite number of blocks and choose one representative value from each block [Ammann and Offutt 2008]. Consider the following 4 blocks to partition the domain of all integers based: b_1: "< 0", b_2: "== 0", b_3: "== 1", b_4: "> 1". Block b_1 contains all integers that are less than 0; b_2 contains just one integer, i.e., 0; b_3 also contains just one integer, i.e., 1; and b_4 contains all integers greater than 1. Next, we pick the following representative values for each block: b_1: -1, b_2: 0, b_3: 1, b_4: 2. Blocks b_2 and b_3 each contain just 1 integer, which must be selected. For blocks b_1 and b_4, we use the *boundary* values, i.e., -1 and 2 respectively. We use the same blocks and representative values for each of the 3 parameters. Now we apply the all combinations coverage (ACoC) criterion to combine the values across different blocks for different parameters. The ACoC criterion requires all combinations of blocks (representative values) from all parameters to be covered. For the chosen blocks, ACoC requires $4 \times 4 \times 4 = 64$ test requirements to be covered. The following is a partial list of test executions explored by ACoC:

```
1: -1, -1, -1 --> 4           ...
2: -1, -1, 0 --> 4            60: 2, 1, 2 --> 2
3: -1, -1, 1 --> 4            61: 2, 2, -1 --> 4
4: -1, -1, 2 --> 4            62: 2, 2, 0 --> 4
5: -1, 0, -1 --> 4            63: 2, 2, 1 --> 2
...                           64: 2, 2, 2 --> 3
```

Each entry shows an id for the test execution, followed by the three parameter values, followed by the result; --> is the separator between the input values and the output.

2.2 Our approach

Observe that while each of the 64 test inputs created by traditional all combinations coverage is unique in terms of the specific combination of input values it tries, not all test inputs exercise different program behaviors. Consider, for example, tests 1 and 2. Both tests set parameter `Side1` to value -1. When `Side1` is -1, the method `Triang` returns 4 *without* using (reading) the values of parameters `Side2` and `Side3`, which means that the method's execution, when `Side1` is -1, does not depend on the values of `Side2` and `Side3`. Therefore, tests 1 and 2 are equivalent in terms of exercising the method's behavior. Simply executing any one of them covers the behavior and gives the result for the other without executing it at all. This observation forms the basis of our approach to optimize all combinations coverage.

Specifically, our approach executes a test and observes which parameter values are actually used (read) by the method under test, and based on that it creates the next test. To illustrate, for input $\langle -1, -1, -1 \rangle$, the method only uses the first parameter's value and therefore our approach creates the next test which has a different value for that parameter, i.e., $\langle 0, -1, -1 \rangle$. Doing so prunes from the search all (16) combinations of the form $\langle -1, ?, ? \rangle$. Thus the first test that our approach runs serves as a representative for 16 unique combinations, which all exercise the same behavior and result in the same output. In the end, our approach creates the following 22 tests for all combinations coverage:

```
1: -1, -1, -1 --> 4          12: 1, 2, 2 --> 2
2: 0, -1, -1 --> 4           13: 2, -1, -1 --> 4
3: 1, -1, -1 --> 4           14: 2, 0, -1 --> 4
4: 1, 0, -1 --> 4            15: 2, 1, -1 --> 4
5: 1, 1, -1 --> 4            16: 2, 1, 0 --> 4
6: 1, 1, 0 --> 4             17: 2, 1, 1 --> 4
7: 1, 1, 1 --> 3             18: 2, 1, 2 --> 2
8: 1, 1, 2 --> 4             19: 2, 2, -1 --> 4
9: 1, 2, -1 --> 4            20: 2, 2, 0 --> 4
10: 1, 2, 0 --> 4            21: 2, 2, 1 --> 2
11: 1, 2, 1 --> 4            22: 2, 2, 2 --> 3
```

For this example, our approach provides a 2.9X reduction in the number of test requirements while providing the same confidence in the method's correctness.

2.3 Symbolic execution

While traditional combinatorial testing is a black-box approach, our approach optimizes it by observing what parameter values are actually used in the method's execution, and therefore makes use of information that exists at the *white-box* level. Next, we compare our approach with a purely white-box approach, namely *symbolic execution* [King 1976; Clarke 1976], which is a classic program analysis technique. In contrast with traditional execution that runs the program on concrete inputs, symbolic execution runs it on symbolic inputs and maintains a symbolic program state, which consists of algebraic expressions that represent values of program variables expressed in terms of the symbolic input values. Moreover, for conditional statements in the program, symbolic execution considers each of the two possible outcomes for condition evaluation separately. Thus, symbolic execution explores each execution path in the program individually. Because programs with loops or recursion can have an unbounded number of paths, symbolic execution tools often bound the length of each execution path and backtrack when the bound is reached to guarantee that the analysis terminates. For each path (prefix) that symbolic execution explores, it checks the path feasibility by (1) building a *path condition*, i.e., a constraint on input symbols that is a necessary condition for that path to be taken by the inputs; and (2) checking the feasibility of the path condition by using off-the-shelf decision procedures, e.g., satisfiability-modulo-theory (SMT) solvers such as the popular Z3 [de Moura and Bjorner 2008]. The exploration performed by symbolic execution forms an *execution tree* that contains all paths explored.

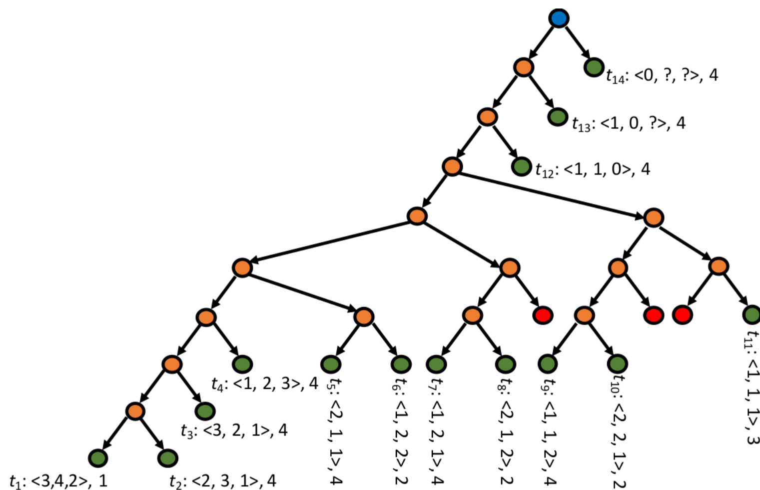

Fig. 1. Symbolic execution tree for `Triang`. Each green leaf node represents a feasible path and is annotated with a test that executes that path; each test contains a triple that represents the input values, and the corresponding output. The specific values in the test inputs created by symbolic execution depend on the backend constraint solver it uses; an input value of "?" indicates that any integer value can be used because the corresponding symbol is not a part of the path condition, i.e, not used along that path. Each red leaf node represents an infeasible path.

We apply symbolic execution to the `Triang` method to create a set of tests that give full path coverage. Figure 1 shows the execution tree explored by symbolic execution. Symbolic execution creates 14 tests. In addition, it explores three paths that are infeasible. Because symbolic execution is a white-box approach, it creates fewer tests and gives higher code coverage than either our approach or the traditional all combinations coverage. However, symbolic execution requires a much more expensive analysis that involves building path conditions and solving them. Overall, our approach provides a sweet spot between all combinations coverage and symbolic execution. Our approach requires minimal analysis overhead (just monitoring parameter reads) and, as this example illustrates, can require much fewer test executions than traditional all combinations coverage.

3. RELATED WORK

We briefly review most closely related work on combinatorial testing and Korat.

Combinatorial testing has been widely studied in research literature [Grindal et al. 2005; Nie and Leung 2011] and applied in practice. AETG [Cohen et al. 1997] is one of the most widely used combinatorial testing systems. An important topic in the context of combinatorial testing is constraints among parameters that rule out some combinations of parameter values [Cohen et al. 2007]; our approach is orthogonal to the work on constraints as we propose how to find definitely equivalent combinations, which can be done for the combinations allowed by the constraints. There is also work on reconstructing models (thus implicitly constraints, albeit of a different kind) from the code under test [Bures et al. 2018]; our approach cannot apply in a pure black-box manner on the code but could apply in a white-box manner on the model. There have been also approaches that investigated the interplay between combinatorial testing and symbolic execution (e.g., [Gao et al. 2016; Grieskamp et al. 2009]); however, to the best of our knowledge, no prior approach considered Korat-like search for combinatorial testing.

Korat has been applied for test generation of several software systems, e.g., recently detecting 59 previously unknown bugs in products of eBay and Yahoo! [Zhong et al. 2016a; 2016b]. In fact, the comKorat work [Zhong et al. 2016a; 2016b] is closely related to this paper because comKorat uses ideas from

combinatorial testing to reduce the inputs generated by Korat in its traditional setting of constraint-based test input generation. A more general approach is introduced in field-exhaustive testing [Ponzio et al. 2016] where coverage criteria from combinatorial testing are integrated in constraint-based test generation to reduce the number of tests. Conceptually, this paper takes the opposite direction from comKorat and field-exhaustive testing: we use ideas from Korat to reduce the number of combinations required by combinatorial testing. Similar in this spirit, some prior work used Korat-style pruning to reduce test executions for configurable systems, including an evaluation in an industrial setting [Kim et al. 2013]. The key difference is that the prior work focused only on setting the configuration parameters for a specific class of systems that were constructed to be configurable, and executed existing tests, but did not consider creating new tests from all combinations, as we do in this paper.

4. CONCLUSIONS

Combinatorial testing is widely used to generate test inputs from combinations of parameter values. The strongest combinatorial coverage criterion is all combinations, which gives the most assurance but often has too many test requirements, being impractical to apply. We introduced a simple approach that provides the same guarantees as all combinations coverage but typically with fewer test inputs, thereby reducing the overall cost of combinatorial testing. Being white-box, our approach can perform better than pure black-box testing. We illustrated how our approach builds on the Korat test generation to explore which combinations of parameter values are equivalent with respect to the code under test. An illustration on the pedagogical example of the triangle classification shows how our approach can substantially reduce the number of test inputs. We leave it for future work to evaluate our approach on a larger scale and to further investigate improving combinatorial testing using Korat-like search.

REFERENCES

Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *ISSTA*.

Miroslav Bures, Karel Frajták, and Bestoun S. Ahmed. 2018. Tapir: Automation Support of Exploratory Testing Using Model Reconstruction of the System Under Test. *IEEE Trans. Reliability* 67, 2 (2018).

L. A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE TSE* 2, 3 (1976).

David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatiorial Design. *IEEE TSE* 23, 7 (1997).

Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*.

Leonardo de Moura and Nikolaj Bjorner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

Ruizhi Gao, Linghuan Hu, W. Eric Wong, Han-Lin Lu, and Shih-Kun Huang. 2016. Effective Test Generation for Combinatorial Decision Coverage. In *QRS*.

Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. 2009. Interaction Coverage Meets Path Coverage by SMT Constraint Solving. In *TestCom/FATES*.

Mats Grindal, Jeff Offutt, and Sten F. Andler. 2005. Combination Testing Strategies: A Survey. *STVR* 15, 3 (2005).

Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don S. Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/SIGSOFT FSE*.

James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19, 7 (1976).

Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2 (2011).

Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. 2016. Field-Exhaustive Testing. In *SIGSOFT FSE*.

Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2016a. Combinatorial generation of structurally complex test inputs for commercial software applications. In *SIGSOFT FSE*.

Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2016b. The comKorat Tool: Unified Combinatorial and Constraint-Based Generation of Structurally Complex Tests. In *NFM*.