Practical Amplification of Condition/Decision Test Coverage by Combinatorial Testing

Artur Andrzejak
Institute of Computer Science
Heidelberg University, Germany
artur.andrzejak@informatik.uni-heidelberg.de

Thomas Bach
Institute of Computer Science
Heidelberg University, Germany
thomas.bach@stud.uni-heidelberg.de

***Abstract*—Test suites in complex software projects might grow over time to considerable sizes, incurring high maintenance effort and prolonged execution times. Maintaining their quality and efficiency require pruning of redundancies while increasing, or at least retaining their coverage levels.**

We propose a lightweight method to tackle these problems with focus on condition/decision coverage (C/D coverage). First, we describe a method to reduce the size of unit tests suites while preserving the degree of their C/D coverage. We then introduce an approach which combines combinatorial testing and input space modeling to further increase the degree of the C/D coverage.

Our semi-automated method works even in absence of models or documentation, and it produces a low number of new test cases requiring queries to a test oracle. We also do not use symbolic execution techniques due to their complexity and limited tool availability for some languages. These properties make our approach practically applicable in industrial projects, and simpler to implement.

We evaluate our approach on selected examples from SAP HANA, a very large industrial application in C++. We demonstrate that it is possible to generate from integration tests new suites of unit tests with high C/D-coverage but with only few test cases. At the same time, the human effort of creating such suites is moderate.

***Keywords*—Unit Testing; Combinatorial Testing; Input Space Modeling; Test Coverage; Test Suite Minimization**

I. Introduction

One of the most important aspects for software projects is software quality. Software quality can have multiple characteristics, but one of the most evident characteristics for end users is the presence or absence of bugs or defects. Software testing aims to reduce the amount of such bugs. Testing cannot prove the absence, but it can indicate the presence of defects. Only after the presence is known, a defect can be fixed and each fixed defect should reduce the probability that end users face a failure during program execution. To ensure that this process of quality improvement is effective for all possible program executions, developers must identify all parts of a software program that are tested and untested. For this purpose, practitioners and researchers have proposed multiple types of code coverage criteria, i.e. metrics that indicate the degree to which the source code of a program is executed by tests.

The purpose of code coverage is widely discussed in literature. One of the opinions says that it is potentially inappropriate to use coverage metrics as a single indicator for quality measurement of a software application. However, coverage or more precisely the absence of coverage indicates untested source code. Untested source code can potentially contain unknown failures that can be detected by corresponding tests.

Researchers have proposed a wide range of techniques to automatically create tests covering untested code. These techniques can be summarized as coverage-adaptive test creation. One family of techniques for coverage-adaptive test creation regards application code as a black-box, and after test creation verifies whether test coverage within the black-box has increased or not. Another option is to analyze the source code as a white-box, and to create suitable tests which reach uncovered parts of the source code. A popular technique for the latter group utilizes constraint solvers or SAT solvers to determine the inputs that trigger execution of certain parts of the code. However, such approaches have practical limitations due to highly structured inputs, external libraries and large complex program structures. These factors pose significant challenges to obtain solutions in an effective way, and this despite of the NP-completeness of the underlying decision or constraint satisfaction problems.

We propose to use combinatorial testing as a technique for the automatic generation of a small (or even minimal) amount of new tests needed to increase the condition/decision coverage of an existing test suite. Our approach first reduces an existing test suite, i.e. trims the number of test cases in the suite while maintaining the degree of condition/decision coverage. It then collects the input values used in the remaining tests, and uses them as input levels in the combinatorial test generation. In other words, we do not assume the availability of a model or even documentation while applying combinatorial testing.

We further reduce the covering arrays created by combinatorial testing techniques to a minimal (or at least small) test set that enhances the condition/decision coverage of an existing test suite. Based on the assumptions that existing test inputs have some purpose and new combinations of them consist of partly meaningful input, we expect that these newly generated tests have an advantage compared

to purely random tests. In particular, we expect that they can cover untested parts of the source code, because the original tests already contain some domain knowledge about the internal behavior of the system under test.

Our proposed technique is significantly simpler than the approaches based on symbolic or concolic execution. We only require test inputs and coverage data for the test execution. This must be contrasted with a complex analysis of Boolean conditions and decision and a precise syntactic understanding of the source code which is necessary for symbolic or concolic execution.

We show the results of a preliminary evaluation of our approach on two examples from SAP HANA, a large industrial database application by SAP. The evaluation indicates that our technique can improve existing test suites in terms of condition/decision coverage. However, it is not able to create full coverage in all scenarios. For such cases, we revert to manual design of input levels with additional insights from previous steps of our technique.

Our contributions are the following ones:

- A method for improving the level of condition/decision coverage of an existing test suite via combinatorial testing, while maintaining small suite size and requiring only few test oracle queries. Compared to symbolic execution, our approach has lower technical requirements, and is independent of the source code of the system under test.
- A preliminary evaluation of this approach for condition/decision coverage on selected functions from SAP HANA, a very large industrial application written in C++.

Cover.	Expr.	U_1	U_2	U_3	U_4	U_5	U_6	…	U_{171}
✓	*D1, f*	**1**	**1**			1		…	1
✓	*D1, t*			1	**1**		**1**	…	
✓	*D2, f*	**1**		-	**-**	1	**-**	…	
✓	*D2, t*		**1**	-	**-**		**-**	…	1
✓	*D3, f*	**-**	**-**			-	**1**	…	-
✓	*D3, t*	**-**	**-**	1	**1**	-		…	-
✓	*C1, f*	**1**	**1**			1		…	1
✓	*C1, t*			1	**1**		**1**	…	
✘	*C2, f*			-	**-**		**-**	…	
✓	*C2, t*	**1**	**1**	-	**-**	1	**-**	…	1
✘	*C3, f*			-	**-**		**-**	…	
✓	*C3, t*	**1**	**1**	-	**-**	1	**-**	…	1
✓	*C4, f*	**1**		-	**-**	1	**-**	…	
✓	*C4, t*		**1**	-	**-**		**-**	…	1
✓	*C5, f*	**-**	**-**			-	**1**	…	-
✓	*C5, t*	**-**	**-**	1	**1**	-		…	-

Figure 1. Illustration of the expression-outcome pairs, their coverage, and a minimum hitting set B_r for the function $f1$ in Listing 1. A row X, y corresponds to an expression-outcome pair $X = y$, and a column U_i to a unit test from B. An entry 1 for row X, y and column U_i means that a condition or decision X is executed and evaluated to y under test U_i. Column "Cover." indicates whether test suite covers (✓) the expression-outcome pair $X = y$, or not (✘). The set $B_r = \{U_1, U_2, U_4, U_6\}$ is a minimal hitting set.

II. Methodology

In this section, we first describe a method for reducing the size of unit test suites while maintaining their degree of condition/decision coverage (Section II-B). In Section II-C we then show how the level of the condition/decision coverage can be increased by exploiting combinatorial testing and input space modeling. Figure 2 gives an overview of the complete approach.

A. Definitions and Conventions

In the following we assume that we start with a given suite of unit tests for a target function f. By analyzing (or instrumenting) these tests we can collect for each execution of a target function f the values of all arguments passed to f, and the return value of f. We call such a record an *input/output pair*, and call the combination of input values an *input tuple*.

We analyze these input/output pairs and retain only those which are unique in terms of the input tuples (i.e. after this step each input tuple for f occurs only once). Since the considered functions are deterministic, this is equivalent to computing unique elements over the input/output pairs. We write $B(f)$ or just B for the resulting set of pairs.

Note that each element of B completely characterizes a unit test for f (including the expected test result). In the following, we use the notions input/output pair and unit test interchangeably for the elements of B.

A Boolean expression in any branch or a loop condition of the considered code is called a *decision*. A *condition* is a leaf-level Boolean expression that cannot be broken down into simpler Boolean expressions. Thus, a decision is either a condition, or it consists of several conditions and Boolean operators. For example, a decision like `y < y1 || y == y1` gives rise to two conditions: `y < y1` and `y == y1`. Listing 1 and Table I illustrate this further. For example, decision $D2$ (line 5) generates conditions $C2$, $C3$, and $C4$.

We call a combination of a decision or a condition X with an outcome y (i.e. *true* or *false*) a *expression-outcome pair* and write $X = y$ for it. If a test case executes a decision (or condition) X *and* the outcome of the evaluation is y, we say that this test case *covers* the expression-outcome pair $X = y$. In Figure 1, test U_1 covers $D1 = f$, $D2 = f$, and four other expression-outcome pairs.

For a set S of tests and an expression-outcome pair $X = y$ let $cov_{X=y}(S)$ be a set of those tests in S which

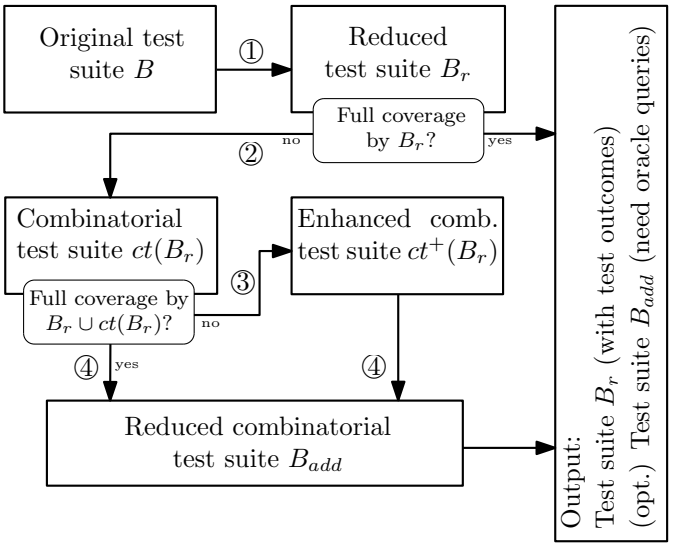

Figure 2. Overview of the processing flow of the complete approach. The major processing steps are:
①: Reducing the original test suite B by solving a hitting set problem (Section II-B). ②: Generating combinatorial tests over B_r using projections of input tuples (Section II-C1). ③: Generating combinatorial tests with support of input space modeling (Section II-C2). ④: Reducing a combinatorial test suite (Section II-C3).

cover $X = y$. For example, for a condition $C1$ with outcome *true* the collection $cov_{C1=t}(S)$ contains all tests from S which execute $C1$ and evaluate it to *true*. In Figure 1, the set $cov_{D1=f}(B)$ are all the unit tests U_i (columns) which have a 1 in their first row.

A test suite achieves a full *condition/decision coverage* if (while executing all test cases in the suite) "*every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once*" [1].

B. Test Suite Reduction with Unchanged C/D-Coverage

The purpose of the following approach is to identify a minimal (or at least a small) subset of unit tests from B which (cumulatively) have the same condition/decision coverage (C/D coverage) as B.

Our approach works in the following steps:

1) We first identify all decisions $D_1, \ldots, D_k$ in the target function f, and split each of these decisions into conditions (designated as $C_1, \ldots, C_l$). Currently this step is performed manually but it could be automated by using a parser.
2) For each condition and each decision, we instrument the source code to collect data whether this expression has been evaluated, and if yes, what was the Boolean result. This step is only necessary to compensate that our dynamic code analysis tool *DynamoRio* is currently not configured to collect condition and decision coverage data.
3) For each input/output pair in B we generate a corresponding unit test and subsequently execute it. At each execution, we collect data for all conditions and decisions described in the previous step.
4) For each expression-outcome pair $X = y$ in the target function f we identify from data generated in the previous step all unit tests in B which cover $X = y$. In other words, we compute the sets $cov_{X=y}(B)$ over all expression-outcome pairs in f.
   The outcome of this step is a collection of $k \leq 2p$ sets of tests, where p is the total number of identified decisions and conditions in a target function f. If any of the computed sets is empty (i.e. $k < 2p$), the corresponding expression-outcome pair is not covered. Figure 1 illustrates that for Example 1 (Section III-B1) the original unit tests have already covered $k = 14$ out of $2p = 16$ expression-outcome pairs, and thus only two sets $cov_{C2=f}(B)$ and $cov_{C3=f}(B)$ are empty.
5) Finally, for the test suite reduction we compute an exact or approximate *minimal hitting set* B_r [2]. Here B is the universe, and the subsets to be "hit" are the $S_1, \ldots, S_k$ (non-empty) sets $cov_{X=y}(B)$ computed in the previous step. By the definition of the minimal hitting set problem, B_r has at least one element in each subset S_i (this element "hits" S_i), and has the smallest size among all subsets of B. In other words, a test suite induced by B_r is guaranteed to have the same C/D-coverage as the full test suite B. Typically, B_r has substantially less elements than B.
   In Figure 1, set $B_r = \{U_1, U_2, U_4, U_6\}$ is a minimal hitting set for the universe B and the sets to be hit being all non-empty sets $cov_{X=y}(B)$ (i.e. all except for $cov_{C2=f}(B)$ and $cov_{C3=f}(B)$).

To compute the minimal hitting set we convert the problem to an equivalent minimum set cover problem [3], and use a proprietary solver implementation introduced in [4]. The latter solver provides exact solution for problem instances with simple structure, or for sizes with approximately $|2p| \leq 20$, and a heuristic solution for larger problems.

The resulting test suite B_r is possibly not achieving the full C/D-coverage, if B was not fully covering. To increase the degree of coverage we apply in the following combinatorial testing enhanced by input space modeling.

C. Improving C/D-Coverage via Combinatorial Testing

For a set S of input/output pairs, let $P_i(S)$ denote the set of all unique values encountered as the i-th component of any input tuple from S. In other words, such a *projection on i-th argument* is the set of all different values used as i-th input argument for a target function f while executing test suite corresponding to S. For example, if f has 3 arguments x, y, z and S contains the input/output pairs $((1, 2, 3), 6)$, $((5, 2, 1), 8)$, and $((10, 3, 7), 20)$, the projection $P_1(S)$ is $\{1, 5, 10\}$ and the projection $P_2(S)$ is $\{2, 3\}$.

By examining the original B or the reduced test suites B_r we observed that expression-outcome pairs $X = y$ are

more frequently not covered if X is a condition and not a decision (the higher decision coverage can be attributed to the high line coverage in our samples). Among the uncovered expression-outcome pairs with conditions we identified two types:

A. A "complex" condition like `b <= (to - from + 1)`, where two or more input arguments are involved directly or indirectly (i.e. if one of the variables is derived from arguments but not an argument itself). In such a case, there is no coverage because in B_r there is no suitable *combination* of input values for involved arguments. Our approach (Section II-C1) attempts to solve this issue by using combinatorial testing to create more combinations of values for these arguments. For each argument, we use as values only the projections of B_r (i.e. original values for this argument). Note that this might not be sufficient, making it necessary to proceed as in Section II-C2.

B. A "simple" condition like `variable == val` (or e.g. `variable < val`), where `variable` is a function argument or a variable derived from a single function argument. In such a case the condition is not fully covered because there is no appropriate value in the projection of B_r on the argument relevant for `variable`. In this case, combinatorial testing based on projections of B_r cannot help, and we need to introduce new values for the relevant argument prior to combinatorial test generation (Section II-C2).

1) Generating combinatorial tests from projections: Our scenario does not assume knowledge about a model for a target function f, which makes the traditional model-based input space modeling not applicable. Instead, we derive levels for each argument of f (i.e. concrete values for this argument used in the combinatorial test suite generation) as follows: levels for i-th argument are exactly all different values used for this argument in the test set S. In other words, the levels for the i-th argument of f are the projections $p_i(S)$ of S on i-th argument.

We then check whether there are any uncovered conditions of type A., and for each such a condition we determine the number m of involved input arguments. For the subsequent test generation, we then set the interaction level t as maximum over all uncovered conditions of type A. To obtain a combinatorial test suite, we generate a covering array by a tool like *Advanced Combinatorial Testing System* (ACTS) [5] with a pre-computed interaction level t.

Note that this method works with the original set B of all logged unit tests, and the reduced suite B_r maintaining the original C/D-coverage level (Section II-B). Since B is likely to have larger projections per argument, we only generate a combinatorial test suite over B_r. Given a fixed interaction level t, we write $ct_t(B_r)$ or just $ct(B_r)$ for the set of combinatorial unit tests generated in this step.

In the subsequent step, we check the C/D coverage of the generated combinatorial suite (as described in Section II-B, Step 4). If some conditions or decisions are still not covered, we additionally execute the steps described in the next subsection.

2) Input modeling for combinatorial tests: To improve the coverage of uncovered decisions or conditions of type A. or B., it might be necessary to perform manual input space modeling prior to generating a combinatorial test suite. We collect the identical levels $p_i(S)$ as in Section II-C1. Then, for each uncovered condition of type B. (e.g. `variable == val`) we simply add a level which evaluates this condition to the missing *true/false* outcome.

For conditions of type A. (multiple variables) we consider the projections for all involved input arguments, and then add level(s) such that at least one combination (of original and new levels) evaluates the condition to a missing *true/false* value. We also set the minimum interaction level t to the number of involved input arguments. A similar approach is taken for the decisions.

Overall, there is no guarantee that suitable levels will be found, and this process is highly manual. In future work, we will use constraint satisfaction methods to automatize this part.

After the levels for all arguments are fixed, we generate a combinatorial test suite (with pre-specified interaction level t) similarly as in Section II-C1. We designate it by $ct_t^+(B_r)$, or just by $ct^+(B_r)$.

3) Reducing combinatorial test suites: A combinatorial test suite T (either $ct(B_r)$ by method in Section II-C1 or $ct^+(B_r)$ by method from Section II-C2) typically has significantly more test cases than the original set B_r. Moreover, most of the combinatorial unit tests have input combinations not in B, and thus require queries to a test oracle.

For these reasons, we apply again a test suite reduction technique described in Section II-B, Step 5. However, the universe is now the set of input tuples corresponding to the tests in the CT test suite T, and the sets $S_1, \ldots, S_k$ to be "hit" correspond *only* to the conditions and decisions not covered (or "hit") by the reduced test suite B_r prior to combinatorial test generation.

For example, assume that f has three decisions $D_1, \ldots, D_3$ and five conditions $C_1, \ldots, C_5$, and B_r (or, equivalently B) does not cover neither of the two cases $C2 = false$ and $C3 = false$. We would perform here the test suite reduction only with $cov_{C2,false}(T)$ and $cov_{C3,false}(T)$ (i.e. computed over T) as sets to be hit. This ensures that the resulting reduced test suite (B_{add}, say) contains only unit tests which handle condition/decision and outcome combinations which are *not yet covered* by B_r.

The overall output of our approach is a test suite with all the tests from B_r, and additional tests from the just reduced suite B_{add}. The earlier tests cover the same expression-outcome cases as B, while latter attempt to

```
1 size_t f1(size_t from,size_t to,size_t a,size_t b){
2     if (from == 1) {
3         return (b < to) ? b : a;
4     }
5     if (b>0 && b <= (to-from+1) && (a < from+b)) {
6         return from + b;
7     }
8     return a;
9 }
```

Listing 1. Source code of Example 1, partially obfuscated

C/D Name	Expression	C/D Name	Expression
$D1$	(Line 2)	$C2$	`b > 0`
$D2$	(Line 5)	$C3$	`b <= (to - from + 1)`
$D3$	(Line 3)	$C4$	`a < from + b`
$C1$	`from == 1`	$C5$	`b < to`

Table I
CONDITIONS AND DECISIONS IN EXAMPLE 1; "(LINE k)" INDICATES THAT THE DECISION CAN BE FOUND IN LINE k OF LISTING 1.

cover additional cases. Note that we need to query the test oracle for all new tests in B_{add}, but not those from B_r.

III. EXPERIMENTS AND EVALUATION

A. System Under Study and Test Environment

Our evaluation is based on selected example functions of SAP HANA, a database management system developed by SAP. SAP HANA consists of several million lines of source code, mainly written in C and C++. The code basis combines and integrates several sub-projects with a lifetime of more than 10 years. We provide details about the testing practices at SAP HANA and gaps we identified between research and practice in our previous work [4]. In particular, we found that several research tools could not be applied to SAP HANA because they are either written for other programming languages (mainly Java) or they do not support very large projects with all variants of the C++ ISO standard features. For example, code coverage generation for the main suite of integration and regression tests requires more than 200 hours if executed sequentially. These low numbers can only be reached due to the restriction to line coverage. Regular branch or statement coverage is economically not feasible. Due to the limitations caused by programming languages and project size, language independent tools and techniques with low requirements in terms of runtimes, memory consumption and manual effort are of high interest for large industrial applications as SAP HANA.

Due to the large size of SAP HANA, we have a large set of tests we can use to extract input values for our approach. We have investigated several approaches to extract inputs from the existing set of tests, for the purpose of this paper we just assume that we mine input arguments from source code of existing unit tests. In fact, we could utilize all approaches for input generation that create some meaningful input. E.g., we could also ask developers to provide meaningful input or conclude equivalence classes and representatives from a model.

Phase	**a**	**b**	**c**	**d**	**e**
Suite	B	B_r	$ct(B_r) \cup B_r$	$ct^+(B_r) \cup B_r$	B_{add}
#tests	171	4	36	38	2
#no oracle	0	0	32	34	2
#missing	2	2	1	0	0
$D1$	152/19	2/2	28/8	5/33	-
$D2$	6/13	1/1	4/4	10/23	-
$D3$	131/21	1/1	22/6	4/1	-
$C1$	152/19	2/2	28/8	5/33	-
$C2$	19/**0**	2/**0**	8/**0**	27/4	1/1
$C3$	19/**0**	2/**0**	5/3	17/12	0/1
$C4$	6/13	1/1	4/1	10/7	-
$C5$	131/21	1/1	22/6	4/1	-

Table II
INTERMEDIATE RESULTS OF THE APPROACH FOR EXAMPLE 1 FOR THE PHASES A – E DESCRIBED IN TEXT. ROW *Suite* SHOWS THE USED TEST SUITE, *#tests* STATES THE NUMBER OF TEST CASES IN THE SUITE, *#no oracle* GIVES THE NUMBER OF TESTS WHICH NEED AN ORACLE QUERY, *#missing* STATES THE NUMBER OF UNCOVERED EXPRESSION-OUTCOME PAIRS. EACH OF THE REMAINING ROWS CORRESPONDS TO A DECISION OR A CONDITION X, AND FOR EACH PHASE THE ENTRY p/q SAYS THAT THE SUITE HAD p TESTS WHICH COVERED THE EXPRESSION-OUTCOME PAIR $X = true$ (I.E. $|cov_{X=t}| = p$), AND THERE WERE q TESTS WHICH COVERED THE EXPRESSION-OUTCOME PAIR $X = false$ (I.E. $|cov_{X=f}| = q$).

B. Improvements of C/D-coverage

We evaluated our approach on several examples from the SAP HANA suite, and discuss here two of them. For reasons of confidentiality, we have partially obfuscated the source code and the variable names.

1) Example 1: Example 1 is a function $f1$ shown in Listing 1. We have extracted from the source code three decisions and five conditions shown in Table I.

Table II shows intermediate results after five essential phases of the algorithm. Phase a refers to analyzing the original tests and the corresponding column shows statistics for the test suite B. We have here 171 unit tests but two expression-outcome cases are not covered, namely $C2 = false$ and $C3 = false$ (see e.g. row $C2$, where only 19 from 171 unit tests execute this condition, and all outcomes are *true*).

Phase b refers to the first reduction (Section II-B) and shows numbers for B_r. The four tests in this suite have the same C/D-coverage as B. In particular, both $C2 = false$ and $C3 = false$ are not covered.

Phase c is the combinatorial test generation from projections (Section II-C1). We show here the results for the union $ct(B_r) \cup B_r$ as adding tests from B_r ensure that this step does not reduce the total coverage (the generated suite of the combinatorial tests might not contain the tests from which projection is taken). In this step, combinatorial testing indeed manages to increase the C/D-coverage. Out of 32 tests found by combinatorial testing for 100% 2-way coverage, 3 tests cover $C3 = false$.

Phase d refers to input modeling for combinatorial tests (Section II-C2). We show here statistics for $ct^+(B_r) \cup B_r$. By adding a new level 0 for argument b we can cover $C2 = false$ (since condition $C2$ is "`b > 0`"). After this phase,

all conditions and decisions are covered, but there are too many (combinatorial) tests - 34 new tests.

In phase e we reduce these CT-generated new tests (i.e. $ct^+(B_r)$) as described in Section II-C3, but only for the following two sets to be hit: $cov_{C2=f}(B_r)$ and $cov_{C3=f}(B_r)$. The result is a test suite B_{add} with two test cases, which covers both $C2 = false$ and $C3 = false$.

Note that the final test suite is comprised of 4+2 test cases, 4 (in B_r) already with expected test results, and 2 (in B_{add}) which require queries to test oracle.

Summary. In this case, the derived unit tests already produced a high C/D-coverage, with 14 of 16 expression-outcome pairs covered. The reduced test suite featured the same C/D-coverage but had only 4 (down from 171) unit tests. "Basic" combinatorial testing (Section II-C1) could cover one more expression-outcome pairs, and further input modeling (Section II-C2) allowed covering of the last open case. We conclude that both coverage enhancement via "basic" combinatorial testing as well as in combination with input modeling are useful in this case, and offer a trade-off between degree of human involvement (for input modeling) and level of the C/D-coverage.

2) Example 2: Example 2 is a function $f2$ that compares two date data and which essentially consists of a single decision using short circuit evaluation (Listing 2). We have extracted from this decision nine conditions shown in Table III.

Table IV shows intermediate results for $f2$ (same format as Table II). The extracted unit tests (i.e. B) cover all expression-outcome pairs except for $C9 = true$. Unfortunately, none of the two variants of coverage enhancement was successful in this case. On the other hand, there is an input tuple (unit test) which covers this case, which we have found manually (see below).

We discuss in more detail the input modeling step (Section II-C2). With the levels for `mi1` and `mi2` derived from B_r it is not possible to evaluate condition $C9$ (`mi1 < mi2`) to true, as all levels for `mi1` are larger 0 and `mi2` is always 0.

Therefore, we added 10 as a new level for `mi2` to the input set for combinatorial testing. We calculated new result sets for 100% 2-way coverage and 100% 4-way coverage. Both result sets did not cover $C9 = true$. We expect that (at latest) a test suite with a full 10-way coverage should contain a suitable input, but the tool which we use did not allowed interaction level t larger than 6 (and could output at most 10 000 test cases).

Instead, we found manually an input tuple which satisfies $C9 = true$ by modifying the (unique) input tuple in B_r which covers $C9 = false$ (namely $2008, 10, 5, 2, 5, 2008, 10, 5, 2, 0$): we replaced here the last argument value 0 by 10 (any value larger 5 would work). Obviously, such cases can be solved by a constraint solver, by seeding it with the additional information we already have from execution. We will investigate this option in our future work.

```
1 bool f2(int y1,int m1,int d1,int h1,int mi1,
2         int y2,int m2,int d2,int h2,int mi2) {
3     return y1 < y2 ||
4            (y1 == y2 && (m1 < m2 ||
5            (m1 == m2 && (d1 < d2 ||
6            (d1 == d2 && (h1 < h2 ||
7            (h1 == h2 && mi1 < mi2)))))));
8 }
```

Listing 2. Source code of Example 2, partially obfuscated

C/D Name	Expression	C/D Name	Expression
$D1$	(Lines 3-7)	$C5$	`d1 < d2`
$C1$	`y1 < y2`	$C6$	`d1 == d2`
$C2$	`y1 == y2`	$C7$	`h1 < h2`
$C3$	`m1 < m2`	$C8$	`h1 == h2`
$C4$	`m1 == m2`	$C9$	`mi1 < mi2`

Table III
CONDITIONS AND DECISIONS IN EXAMPLE 2. DECISION $D1$ CAN BE FOUND IN LINES 4-8 OF LISTING 2.

IV. Related Work

Our work is broadly related to test amplification, symbolic execution, test coverage, and combinatorial testing.

Test amplification describes a set of approaches targeting improving test quality, see a recent survey by Danglot et al. [6]. Related terms found in the literature are test augmentation and test enhancement.

Our approach creates new tests to maximize condition/decision coverage and can be interpreted as an alternative to symbolic execution. Orso and Rothermel report the latter technique as one of the most frequently mentioned major contribution made since 2000. However, despite the excitement and amount of novel research, symbolic execution has practical limitations due to highly structured inputs, external libraries and large complex program structures [7]. Our approach has less limitations for practical applications and utilizes combinatorial testing as an alternative to symbolic execution techniques to tackle complex program structures.

Phase	**a**	**b**	**c**	**d**	**e**
Suite	B	B_r	$ct(B_r) \cup B_r$	$ct^+(B_r) \cup B_r$	B_{add}
#tests	256	9	56	57	0
#no oracle	0	0	47	48	0
#missing	1	1	1	1	1
$D1$	113/143	5/5	34/22	34/22	-
$C1$	12/244	1/8	28/28	28/29	-
$C2$	194/50	7/1	10/18	11/18	-
$C3$	51/143	1/6	3/07	4/07	-
$C4$	96/47	5/1	6/1	6/1	-
$C5$	27/69	1/4	2/4	2/4	-
$C6$	46/23	3/1	3/1	3/1	-
$C7$	23/23	1/2	1/2	1/2	-
$C8$	17/6	1/1	1/1	1/1	-
$C9$	**0**/17	**0**/1	**0**/1	**0**/1	-

Table IV
INTERMEDIATE RESULTS OF THE APPROACH FOR EXAMPLE 2 FOR THE PHASES A – E DESCRIBED IN TEXT. THE VALUES HAVE THE SAME MEANING AS IN TABLE II.

Work of Bloem et al. [8] is probably most closely related to our study. In fact, they apply similar algorithmic steps as our approach. Bloem et al. utilize a set of existing test cases to iteratively create new test cases until a termination criterion is met. They generate a new test case by a backtracking constraint solver that iterates over all possible execution paths until it reaches a desired branch of the source code. The evaluation shows that this approach can create up to 100% branch coverage. However, their approach is applicable only under certain preconditions. For example, changes to source code should be possible, and long execution times acceptable.

Coverage criteria in general and C/D coverage used in our work have the benefit that they can be automatically measured and compared for different approaches, leading to a plethora of studies [9], [10]. However, Staats et al. points out that coverage criteria such as MC/DC coverage can be ineffective for determining test suite adequacy [11]. Inozemtseva et al. conclude that coverage is not strongly correlated to test suite effectiveness when test suite size is controlled [12]. In our previous work on the correlation between coverage and bugs in a large industrial software, we have shown that coverage correlates to future bugs [13]. This result supports the argument that it may be beneficial to create tests for uncovered and therefore untested parts of the source code, at least for our system under test.

Several research works analyze combinations between combinatorial testing and coverage. Choi et al. study the code coverage (line, branch) effectiveness of combinatorial$t$-way testing with small t for grep, make and flex in different versions [14]. They found that t-way testing creates already an efficient coverage compared to exhaustive testing for $t < 5$. Czerwonka investigates the effects of combinatorial testing to coverage for several utility programs in MS Windows [15]. Czerwonka concludes that full t-way coverage can lead to test suites with same t-way coverage but different (line, statement) code coverage. We can confirm this observation from our experience with the examples. However, we expect this effect to decrease if the interaction level t becomes comparable to the amount of arguments.

V. Conclusions

We propose an approach to reduce size of existing test suites and enhance them with new tests created by combinatorial testing techniques. Our preliminary evaluation shows that our method can improve the condition/decision coverage for two examples while reducing the size of the test suite. Simultaneously, there are only few tests which require test oracle queries. Our approach has lower technical requirements compared to symbolic or concolic execution. On the other hand, our approach has several limitations. For example, in some scenarios it could not generate additional test cases to achieve a full C/D coverage.

More extensive analysis would be required to further understand the robustness and applicability of the proposed technique. We plan to conduct an evaluation with a larger set of functions. This would increase our understanding of characteristics of functions and original test suites for which our approach produces good results. In addition, our future research will include getting feedback from practitioners about the practical implications of our technique (in form of interviews or surveys). As discussed in Section III, we also see potential benefits of approaches which combine measuring code coverage, combinatorial testing, and symbolic execution.

References

[1] Wikipedia contributors, "Modified condition/decision coverage." https://en.wikipedia.org/w/index.php?title=Modified_condition/decision_coverage&oldid=819591271. [Online; accessed 10-January-2018].

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pp. 85–103, Plenum Press, New York, 1972.

[3] Wikipedia contributors, "Set cover problem." https://en.wikipedia.org/w/index.php?title=Set_cover_problem&oldid=813978581. [Online; accessed 10-January-2018].

[4] T. Bach, A. Andrzejak, and R. Pannemans, "Coverage-based reduction of test execution time: Lessons from a very large industrial project," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 3–12, March 2017.

[5] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, pp. 370–375, Mar. 2013.

[6] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry, "The emerging field of test amplification: A survey," *CoRR*, vol. abs/1705.10692, 2017.

[7] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*, FOSE 2014, (New York, NY, USA), pp. 117–132, ACM, 2014.

[8] R. Bloem, R. Koenighofer, F. Röck, and M. Tautschnig, "Automating test-suite augmentation," in *2014 14th International Conference on Quality Software*, pp. 67–72, Oct 2014.

[9] K. J. Hayhurst and D. S. Veerhusen, "A practical approach to modified condition/decision coverage," in *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, vol. 1, pp. 1B2/1–1B2/10 vol.1, Oct 2001.

[10] T. K. Paul and M. F. Lau, "A systematic literature review on modified condition and decision coverage," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, (New York, NY, USA), pp. 1301–1308, ACM, 2014.

[11] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, (Berlin, Heidelberg), pp. 409–424, Springer-Verlag, 2012.

[12] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 435–445, ACM, 2014.

[13] T. Bach, A. Andrzejak, R. Pannemans, and D. Lo, "The impact of coverage on bug density in a large industrial software project," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 307–313, Nov 2017.

[14] E.-H. Choi, O. Mizuno, and Y. Hu, "Code Coverage Analysis of Combinatorial Testing," in *QuASoQ/TDA@ APSEC*, pp. 43–49, 2016.

[15] J. Czerwonka, "On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Designs," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 257–266, Mar. 2013.