2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation

Linbin Yu[1], Yu Lei[1], Mehra Nourozborazjany[1], Raghu N. Kacker[2], D. Richard Kuhn[2]

[1]*Department of Computer Science and Engineering*
University of Texas at Arlington
Arlington, TX 76019, USA
{linbin.yu, mehra.nourozborazjany}@mavs.uta.edu,
ylei@cse.uta.edu

[2]*Information Technology Laboratory*
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{raghu.kacker, kuhn}@nist.gov

***Abstract*— Combinatorial testing has been shown to be a very effective testing strategy. An important problem in combinatorial testing is dealing with constraints, i.e., restrictions that must be satisfied in order for a test to be valid. In this paper, we present an efficient algorithm, called IPOG-C, for constraint handling in combinatorial testing. Algorithm IPOG-C modifies an existing combinatorial test generation algorithm called IPOG to support constraints. The major contribution of algorithm IPOG-C is that it includes three optimizations to improve the performance of constraint handling. These optimizations can be generalized to other combinatorial test generation algorithms. We implemented algorithm IPOG-C in a combinatorial test generation tool called ACTS. We report experimental results that demonstrate the effectiveness of algorithm IPOG-C. The three optimizations increased the performance by one or two orders of magnitude for most subject systems in our experiments. Furthermore, a comparison of ACTS to three other tools suggests that ACTS can perform significantly better for systems with more complex constraints.**

Keywords-Combinatorial Testing; Constraint Handling; Test Genration;

I. INTRODUCTION

Combinatorial testing (CT) has been shown to be a very effective testing strategy [14] [15] [20]. Given a system with *n* parameters, t-way combinatorial testing requires that all t-way combinations, i.e., all combinations involving *any t* parameter values, be covered by at least one test, where *t* is referred to as test strength and is typically a small number. A widely cited NIST study of several fault databases reports that all the faults in these databases are caused by no more than six factors [14]. If test parameters are modeled properly, t-way testing can expose all the faults involving no more than *t* parameters.

Practical applications often have constraints on how parameter values can be combined in a test [11]. For example, one may want to ensure that a web application can be executed correctly in different web browsers running on different operating systems. Consider that Internet Explorer (or IE) 6.0 or later cannot be executed on MacOS. Thus, if the web browser is IE 6.0 or later, the operating system must not be MacOS. This constraint must be taken into account such that IE 6.0 or later and Mac OS do not appear in the same test.

Constraints must be specified by the user before they are handled during test generation. One approach is to specify constraints as a set of forbidden tuples. A forbidden tuple is a value combination that should not appear in any test. When there are a large number of forbidden tuples, it can be difficult for the user to enumerate them. Alternatively, constraints can be specified as a set of logical expressions. A logical expression describes a condition that must be satisfied by every test. Logical expressions are more concise than explicit enumeration of forbidden tuples. In this paper, we assume that constraints are specified using logical expressions.

A major step in constraint handling is validity check, i.e., checking whether all the constraints are satisfied by a test. One approach to performing this check is to ensure that a test contains no forbidden tuples. This approach needs to maintain the complete list of all the forbidden tuples, which can be expensive when there are a large number of forbidden tuples. Alternatively, we can employ a constraint solver to perform this check. In this approach, we encode the problem of validity check as a constraint satisfaction problem. In this paper we focus on the latter approach, since it avoids maintaining the complete set of forbidden tuples and is thus a more scalable approach.

It is important to note that the way in which validity check is performed is independent from the way in which constraints are specified. For example, a tool called mAETG [6] uses forbidden tuples to specify constraints. Forbidden tuples are converted into a set of Boolean logic expressions, which are then solved by a SAT solver. In contrast, a tool called PICT [8] uses logic expressions to specify constraints. A list of forbidden tuples are first generated from the specified logic expressions and then used to perform validity check during test generation.

Both combinatorial testing and constraint solving are computation-intensive processes. The main challenge of constrained combinatorial test generation is dealing with this complexity. In this paper, we present an efficient algorithm, called IPOG-C, to address this challenge. Algorithm IPOG-C modifies an existing combinatorial test generation algorithm called IPOG [17] and employs a constraint solver to handle constraints. To optimize the performance of constraint handling, algorithm IPOG-C tries to reduce the number of calls to the constraint solver. In case that such a call cannot be avoided, algorithm IPOG-C tries to simplify the solving process as much as possible.

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.35

Specifically, algorithm IPOG-C includes the following three optimizations:

1) *Avoiding unnecessary validity checks on t-way combinations*. A t-way test set must cover all the valid t-way combinations. A t-way combination is valid if it can be covered by at least one valid test. Checking the validity of each t-way combination can be expensive since there often exist a large number of t-way combinations. The key insight in our optimization is that if a test is found valid, then all the combinations covered by this test would be valid, and thus do not have to be explicitly checked.
2) *Checking relevant constraints only*. When we perform a validity check, some constraints may not be relevant and thus do not have to be checked. We use a notion called constraint relation graph to identify groups of constraints that are related to each other, which are then used to identify relevant constraints in a validity check. Algorithm IPOG builds a test set incrementally, i.e., covering one parameter at a time. This incremental framework is leveraged in this optimization to further reduce the number of relevant constraints that have to be involved in a validity check.
3) *Recording the solving history*. This optimization tries to reduce the number of calls to the constraint solver by saving previous solving results. This optimization works together with 2) to maximize reduction in the number of calls to the constraint solver.

For the purpose of evaluation, we implemented algorithm IPOG-C in a combinatorial test generation tool called ACTS. ACTS is freely available to the public [1]. We conducted experiments on a set of real-life and synthesized systems. The experimental results indicate that the three optimizations employed in algorithm IPOG-C increased the performance by one or two orders of magnitude for most subject systems. For example, for a real-life system GCC, the optimizations reduced the number of calls to the constraint solver from 34613 to 631 and the execution time from 683.599 seconds to 1.139 seconds. Furthermore, the optimizations significantly slow down the increase in the number of calls to the constraint solver and the execution time as test strength, number of parameters, domain size, or number of forbidden tuples increases. Finally, a comparison of ACTS to three other tools suggests that ACTS can perform significantly better for systems with more complex constraints.

The rest of this paper is organized as follows. Section II gives a formalization of the constrained combinatorial test generation problem. Section III presents the original IPOG algorithm without constraint support. Section IV presents the new algorithm, i.e., IPOG-C. In particular we discuss the three optimizations. Section V reports some experimental results. Section VI discusses related work. Section VII concludes this paper and discusses future work.

II. PRELIMINARY

In this section, we formally define the problem of constrained combinatorial test generation.

Definition 1 (Parameter) A parameter p is a set of values, i.e., $p = \{v_1, v_2, \ldots, v_p\}$.

Value v for parameter p can be denoted as $p.v$. For ease of notation, we assume that different parameters are disjoint. This implies that each parameter value belongs to a unique parameter. This allows us to refer to a parameter value by itself, i.e., without mentioning which parameter it belongs to.

Definition 2 (Tuple) Let $G = \{p_1, p_2, \ldots, p_m\}$ be a set of parameters. A tuple $\tau = \{v_1, v_2, \ldots, v_m\}$ of G is a set of values where $v_i \in p_i$. That is, $\tau \in p_1 \times p_2 \ldots \times p_m$.

Intuitively, a tuple τ consists of a value v for each parameter p in a given set of parameters. We refer to a tuple of size t as a t-tuple. We also refer to v as the value of p in τ if there is no ambiguity. This effectively overloads the notion of a parameter, which may represent a set of values or may take a particular value, depending on its context. We use $\Pi(\{p_1, \ldots, p_m\})$ to denote $p_1 \times p_2 \ldots \times p_m$.

Definition 3 (SUT) A System Under Test (SUT) $M = <P, C>$ consists of a set $P = \{p_1, p_2, \ldots, p_{|P|}\}$ of parameters, where p_i is a parameter, and a set $C = \{c_1, c_2, \ldots, c_{|C|}\}$ of constraints, where each constraint c_i is a function: $\Pi(P) \rightarrow \{true, false\}$.

We refer to each tuple in $\Pi(P)$ as a test of M. In other words, a test is a special tuple whose size equals the number of parameters in a system. A constraint is a function that maps a test to a Boolean value *true* or *false*.

Definition 4. (Covering) A tuple τ is said to be covered by another tuple τ' if $\tau \subseteq \tau'$.

Note that a tuple is covered by itself. In this paper, we are particularly interested in the case where a tuple is covered by a test.

Definition 5. (Validity) Given a SUT $M = <P, C>$, a tuple τ of M is valid if $\exists\ \tau' \in \Pi(P)$, such that $\tau \subseteq \tau'$, and $\forall c \in C$, $c(\tau') = true$. Otherwise, τ is invalid.

If τ is a test, τ is valid if it satisfies all constraints. If τ is a t-tuple, where $t < |P|$, then τ is valid if there exists at least one valid test τ' that covers τ.

Definition 6. (Constrained T-Way Test Set) Let $M = <P, C>$ be a SUT. Let Σ be the set of all valid t-tuples. A t-way constrained test set is a set $\Omega \subseteq \Pi(P)$ of tests such that, $\forall\ \sigma \in \Sigma$, there exists $\tau \in \Omega$ such that τ is valid and $\sigma \subseteq \tau$.

Intuitively, a constrained t-way test set is a set of valid tests in which each valid t-tuple is covered by at least one test. The problem of constrained t-way test generation is to generate a constrained t-way test set of minimal size. In practice, a tradeoff is often made between the size of the resulting test set and the time and space requirements.

III. THE IPOG ALGORITHM

In this section, we introduce the original IPOG algorithm without constraint handling [17]. Due to space limit, we only present the major steps relevant to constraint handling. Refer to the original paper [17] for more details.

Algorithm IPOG works as follows: For a system with t or more parameters, we first build a t-way test set for the first t parameters. We then extend this test set to a t-way test set for the first t+1 parameters, and continue to do so until it builds a t-way test set for all the parameters.

Assume that we already covered the first k parameters. To cover the (k+1)-th parameter, say p, it is sufficient to cover all the t-way combinations involving parameter p and any group of (t-1) parameters among the first k parameters. These combinations are covered in two steps, horizontal growth and vertical growth. Horizontal growth adds a value of p to each existing test. Each value is chosen such that it covers the most uncovered combinations. During vertical growth, the remaining combinations are covered one at a time, either by changing an existing test or by adding a new test. When we add a new test to cover a combination, parameters that are not involved in the combination are given a special value called *don't care*. These *don't care* values can be later changed to cover other combinations.

Fig. 1 illustrates how algorithm IPOG works. Assume that the system contains 4 parameters p1, p2, p3, and p4, and each parameter has 2 values {0, 1}. The test strength is 2. Assume that the 2-way test set for the first 3 parameters has been generated, as shown in Fig. 1(a).

	p1	p2	p3
test 1	0	0	0
test 2	0	1	1
test 3	1	0	1
test 4	1	1	0

(a) The test set for p1, p2 and p3

p1	p4	p2	p4	p3	p4
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
1	1	1	1	1	1

(b) Target tuples for p4

	p1	p2	p3	p4
test 1	0	0	0	**0**
test 2	0	1	1	**1**
test 3	1	0	1	**0**
test 4	1	1	0	**0**

(c) Horizontal growth for p4

	p1	p2	p3	p4
test 1	0	0	0	0
test 2	0	1	1	1
test 3	1	0	1	0
test 4	1	1	0	0
test 5	**1**	**0**	**0**	**1**

(d) Vertical growth for p4

Figure 1. Illustration of the IPOG Algorithm

To cover the last parameter p4, we first generate all 2-way combinations that need to be covered. Fig. 1(b) shows 12 2-way combinations to be covered. During horizontal growth, we add value 0 of P4 into the first test since it covers the most uncovered tuples {p1.0, p4.0}, {p2.0, p4.0} and {p3.0, p4.0}. Similarly, we add values 1, 0 and 0 of P4 into the next three tests, respectively, as shown in Fig. 1(c). There are still 3 uncovered 2-way combinations, {p1.1, p4.1}, {p2.0, p4.1} and {p3.0, p4.1}. During vertical growth, we first generate a new test to cover {p1.1, p4.1}. Then we add p2.0 and p3.0 into the same test to cover {p2.0, p4.1} and {p3.0, p4.1}, respectively. Fig. 1(d) shows the complete 2-way test set.

IV. THE IPOG-C ALGORITHM

In this section, we modify algorithm IPOG to handle constraints. We refer to the new algorithm as IPOG-C. We first present a base version of algorithm IPOG-C. Then we propose three optimizations. The final version of algorithm IPOG-C is obtained by applying these optimizations to the base version. We also discuss how to apply these optimizations to other test generation algorithms.

A. The Base Version of Algorithm IPOG-C

Fig. 2 shows the base version of the IPOG-C algorithm. The modifications made to the original IPOG algorithm are highlighted. These modifications do not change the main structure of the original IPOG algorithm. If no constraints are specified, the modified algorithm will generate the same test set as the original IPOG algorithm does.

Algorithm IPOG-C modifies the original IPOG algorithm to make sure: (1) all the valid t-way target tuples are covered; and (2) all the generated tests are valid. In line 5, we perform validity check on each t-way combination to identify all the valid t-way combinations that need to be covered. In lines 8 & 13, we perform the validity check to ensure that every test is valid. Since the algorithm terminates only when π is empty (line 12), all the valid t-way combinations must be covered upon termination.

```
Algorithm IPOG-C (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set
2. sort the parameters in set ps in a non-increasing order of their
     domain sizes, and denote them as P1, P2, ..., and Pk
3. add into test set ts a test for each valid combination of values
     of the first t parameters
4. for (int i = t + 1; i ≤ k; i ++){
5.    let π be the set of all valid t-way combinations of values
        involving parameter Pi and any group of (t-1) parameters
        among the first i-1 parameters
6.    // horizontal growth for parameter Pi
7.    for (each test τ = (v1, v2, ..., vi-1) in test set ts) {
8.       choose a value vi of Pi and replace τ with τ' = (v1, v2, ...,
           vi-1, vi) so that τ' is valid and it covers the most
           number of combinations of values in π
9.       remove from π the combinations of values covered by τ'
10.   } // end for at line 7
11.   // vertical growth for parameter Pi
12.   for (each combination σ in set π){
13.      if (there exists a test τ in test set ts that can be changed to
           a valid test τ' that covers both τ and σ {
14.         change test τ to τ'
15.      } else {
16.         add a new test only contains σ to cover σ
17.      } // end if at line 13
18.   } // end for at line 12
19. }// end for at line 4
20. return ts;
}
```

Figure 2. The base version of the IPOG-C algorithm

B. Validity Check

Assume that we want to check the validity of a combination (or test) τ for a system S. This validity check problem is converted to a Constraint Satisfaction Problem (CSP), in which the variables are the parameters of S. The constraints include the constraints specified by the user and some constraints derived from τ, where each parameter

value $p.v$ in τ is represented by a constraint expression p = v. (Alternatively, one may change the parameter domain to a fixed value if it is supported by the solver.) A third party constraint solver is then used to solve this CSP.

Consider that a system consists of 3 parameters a, b, c, each having 3 values, and one constraint "a + b > c". Fig. 3 shows the CSP for checking the validity of combination {a.0, b.0}. Note that two constraints "a = 0" and "b = 0" are added for parameter values a.0 and b.0, in addition to the user-specified constraint, i.e., "a + b > c".

[Variable]	[Constraints]
a: 0, 1, 2	(1) a + b > c
b: 0, 1, 2	(2) a = 0
c: 0, 1, 2	(3) b = 0

Figure 3. An example CSP problem

C. Optimizations

In this section, we propose several schemes to optimize the performance of constraint handling in algorithm IPOG-C.

1) Avoiding Unnecessary Validity Checks of Target Combinations

In line 5 of Fig. 2, we first compute the complete set of all valid t-way combinations that need to be covered. This involves performing validity check on each t-way combination. This computation can be very expensive since there are typically a large number of t-way combinations.

We propose an optimization to reduce the number of validity checks on target combinations. The key observation is that there exists significant redundancy between validity checks for finding valid target tuples, and validity checks for choosing a valid parameter value during horizontal growth. That is, when we choose a new value, we perform validity check to ensure that the resulting test is valid. Since all the tuples covered in a test must be valid, this check implicitly checks validity of every tuple covered in this test. As a result, even though we do not have the list of all valid tuples, it is guaranteed that any tuple covered in a test, and removed from the target set π, must be valid.

The above observation suggests the following optimization. That is, we do not need to check the validity of target tuples (line 5 of Fig. 2) before horizontal growth. It is important to note that the existence of invalid tuples in the target set (π) will not affect the greedy selection in horizontal growth (line 8 of Fig. 2). This is because if a candidate test covers any invalid t-tuple, it must be an invalid test and will not be selected. So the effective comparison only happens between valid candidates.

After horizontal growth is finished, validity check needs to be performed on the remaining target combinations. That is, line 12 of Fig. 2 should be changed to "for (each **valid** combination σ in set π)". At this point, many combinations are likely to have already been covered by horizontal growth. This means that the number of validity checks can be significantly reduced.

2) Checking Relevant Constraints Only

For a given validity check, some constraints may not be relevant, and thus do not need to be checked. In this optimization, we identify constraints that are relevant to a validity check and ignore the other ones. This helps to simplify the corresponding constraint solving problem.

We first divide constraints into non-intersecting groups. To do this, we use a graph structure called constraint relation graph, to represent relations between different constraints. In a constraint relation graph, each node represents a constraint, and each (undirected) edge indicates that two constraints have one or more common parameters. Then we find all the connected components in the graph. The constraints in each connected components are put into one group. Intuitively, constraints that share a common parameter, directly or indirectly, are grouped together.

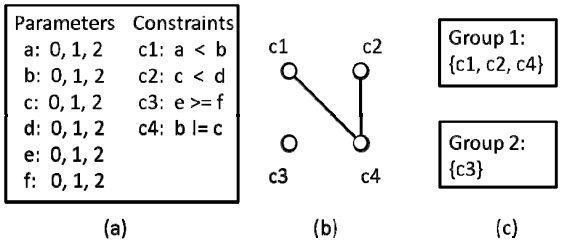

Figure 4. Illustration of Constraint Group

Fig. 4(a) shows a system that contains 7 parameters and 4 constraints. Fig. 4 (b) shows the constraint relation graph for the system. There are two connected components. Fig. 4(c) shows two constraint groups identified from the constraint relation graph. Note that this process only needs to be executed once. As mentioned in Section VI, similar techniques, which are often referred to as slicing, are used inside many constraint solvers.

Now we explain how to use constraint groups to identify irrelevant constraints. To check the validity of a test (or combination) τ, we identify relevant constraints as follows. For each parameter in τ, if it is involved in a constraint c, all the constraints in the same constraint group as c are identified to be relevant to this validity check. Only these constraints are encoded in the CSP sent to the constraint solver.

This optimization can be very effective considering that algorithm IPOG-C builds a test set incrementally. That is, we build a t-way test set for the first t parameters, and then extend this test set to cover the first t+1 parameters, and so on. When we try to cover a new parameter p, we only need to check the constraints in the same group as p.

Consider the example system in Fig. 4(a). Assume we add a new parameter value f.0 to an existing test {a.0, b.1, c.0, d.0, e.1}, which must be valid. To check the validity of the new test, i.e., {a.0, b.1, c.0, d.0, e.1, f.0}, we only need to consider c3. Constraints c1, c2 and c4 are not relevant in this case.

3) Recording the Solving History

In this optimization, we record the solving history for each constraint group to avoid solving the same CSP multiple times. As discussed in IV.B, a CSP is encoded from a set of parameter values in the test that needs to be checked, and is then solved by a constraint solver. A Boolean value "true" or "false" that indicates the validity of

the test will be returned. For each constraint solving call, we save solving history, i.e., the set of parameters values send to the CSP solver and the Boolean value returned by the CSP solver. Next time when a constraint solving call is going to make, we first search for the same set of parameters values in the solving history, and if a match is found, we can reused the cached result to avoid this solving call. Recall that in the previous optimization, we divide constraints into several non-intersecting groups. To increase the hit rate of the cached solving history, we divide a CSP problem into several independent sub-problems based on constraint groups, and then save the solving history for each of them.

Consider the example system in Fig. 4(a). Assume that parameters a, b, c, d and e have been covered and we are trying to cover parameter f. Fig. 5(a) shows 2 different candidate tests. As discussed earlier, the only relevant constraint is c3, which involves parameters e and f. Therefore the validity of the two candidate tests is essentially determined by the the combinations of values of parameters e and f in the two tests. Whereas the 2 tests that need to be checked are different, their validity is determined by the same value combination, i.e. {e.0, f.1}. Thus after checking the first test, we have the solving history for {e.0, f.1} (which is invalid), as shown in Fig. 5(b). This allows us to derive that the second test is invalid without making a call to the constraint solver.

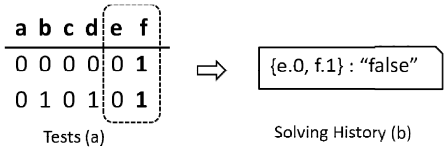

Figure 5. Illustration of using constraint solving history

D. Applying Optimizations to Other Algorithms

Our optimizations can be applied to other test generation algorithms. Due to space limitations, we discuss how to apply our optimizations to the AETG algorithm [5]. Like algorithm IPOG, the AETG algorithm adopts a greedy framework. However, it builds a test set one-test-at-a-time, instead of one-parameter-at-a-time.

The key to apply the first optimization is dealing with the existence of invalid t-way combinations in the target set, which is supposed to contain only valid t-way combinations. This could affect the greedy selection of a test value. In AETG, a test value is selected such that it covers the most valid combinations in the target set. According to the principle of the first optimization, all the combinations covered by a valid test are guaranteed to be valid. Thus, as long as the validity of the resulting test is checked after the selection of a test value, the existence of invalid combinations would not affect the selection process. However, there is an exception with the selection of the first t – 1 values in a test. In the AETG algorithm, these values are selected such that they appear in the most number of combinations in the target set.

One approach to dealing with this exception is to change the AETG algorithm as follows. Instead of choosing the first t – 1 values one by one, we choose the first t values in a test altogether. This is done by finding the first valid t-way combination that remains in the target set, and then assign the t values in this combination to the test. This change makes the selection of the first t values less greedy. However, we note that t is typically small, and the selection of the other values remains unchanged.

The existence of invalid t-way combinations in the target set may also affect the termination condition. In the AETG algorithm, the test generation process terminates when the target set is empty. However, some invalid t-way combinations may never be covered, and thus the target set may never be empty. It is interesting to note that this problem can be resolved by the same change suggested earlier. That is, if we select the first t values of a test altogether by finding a valid t-way combination in the target set, the test generation process comes to a natural stop when no valid combination can be found.

The second and third optimizations do not interact with the core test generation process. That is, they take effect only during valid checks. Thus, they can be applied to the AETG algorithm without modifications.

V. EXPERIMENTS

We implemented algorithm IPOG-C and integrated it into a combinatorial test generation tool called ACTS [1], which is freely available to the public. An open source CSP solver called Choco [4] is used for constraint solving.

Our experiments consist of three parts. The first part (Section V.B) is designed to evaluate the effectiveness of the three optimizations. The second part (Section V.C) is to investigate how the performance of algorithm IPOG-C is affected by several factors, including test strength, number of parameters, size of domain and the number of forbidden tuples. The third part (Section V.D) compares ACTS with other combinatorial test generation tools. All these experiments were performed on a laptop with Core i5 2410M 2.30GHz CPU and 4.0 GB memory, running 64-bit Windows 7.

A. Subject Systems

We use both real-life and synthesized systems in our experiments. The real-life systems include the five systems introduced in [7], and a system called TCAS introduced in [16]. We adopt the exponential notation in [7] to denote parameter configurations, where d^n means that there are n parameters with domain size d. The constraints in these systems were given in the form of forbidden tuples in [7] and [16]. We also use an exponential notation to denote constraints, where d^n means there are n forbidden tuples each of which involves d parameters. We manually convert each forbidden tuple to an equivalent constraint expression. For example, a forbidden tuple {a.0, b.1} is converted to a logic expression "! (a=0 && b=1)". The configurations of 6 real-life systems are listed in Table 1.

We created 10 synthesized systems, all of which consist of 10 parameters of domain size 4. We denote the parameters as p1, p2, …, and p10. Each system contains a single constraint which is carefully designed to control the number of forbidden tuples. The number of forbidden tuples is an important measure of the complexity of a constraint.

Note that the number of constraints is not important as different constraints can be joined together.

TABLE I. CONFIGURATIONS OF REAL-LIFE SYSTEMS

Name	Num. of Parameters	Num. of Constraints	Parameter Configuration	Constraint Configuration
Apache	172	7	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$
Bugzilla	52	5	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$
GCC	199	40	$2^{189}3^{10}$	$2^{37}3^{3}$
SPIN-S	18	13	$2^{13}4^{5}$	2^{13}
SPIN-V	55	49	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$
TCAS	12	3	$2^{7}3^{2}4^{1}10^{2}$	2^{3}

Some existing tools only support forbidden tuples as constraints. To compare to these tools, we derive all the forbidden tuples encoded by each constraint. Take system C1 as an example, we enumerate all 3-way value combinations of parameter p1, p2 and p3, and found 30 combinations that violate the constraint (p1>p2 || p3>p2) as forbidden tuples. We list the configurations, the number of derived forbidden tuples and detailed constraints for these synthesized systems in Table II.

TABLE II. CONFIGURATIONS OF SYNTHESIZED SYSTEMS

Name	Param. Config.	Num. of Forbidd. Tuples	Constraint
C1	4^{10}	30	p1>p2 || p3>p2
C2	4^{10}	100	p1>p2 || p3>p4
C3	4^{10}	200	p1>p2 || p3>p4 || p5>p1
C4	4^{10}	300	p1>p2 || p3>p4 || p5>p2
C5	4^{10}	1000	p1>p2 || p3>p4 || p5>p6
C6	4^{10}	2000	p1>p2 || p3>p4 || p5>p6 || p7>p1
C7	4^{10}	3000	p1>p2 || p3>p4 || p5>p6 || p7>p2
C8	4^{10}	10000	p1>p2 || p3>p4 || p5>p6 || p7>p8
C9	4^{10}	20000	p1>p2 || p3>p4 || p5>p6 || p7>p8 || p9>p1
C10	4^{10}	30000	p1>p2 || p3>p4 || p5>p6 || p7>p8 || p9>p2

B. Evaluation of the Optimizations

To evaluate he effectiveness of individual optimizations and their combination, we tested multiple configurations. Table III shows five different configurations of the optimization options, where a tick denotes that the corresponding optimization is enabled, and dash means not. The first configuration represents the base version of IPOG-C, i.e., without any optimization, and the last one contains all the optimizations.

TABLE III. CONFIGURATION OF OPTIMIZATION OPTIONS

Optimization	Base	O1	O2	O3	All
Avoiding unnecessary validity checks on t-way combinations	-	✓	-	-	✓
Checking relevant constraints only	-	-	✓	-	✓
Recording the solving history	-	-	-	✓	✓

Due to limited space, we use 6 real-life systems and the synthesized system with the most complex constraint, i.e., C10. The test strength is set to 2. We measure the performance of constrained test generation in terms of number of constraint solving calls (i.e., the number of times the constraint solver is called) and execution time. The number of constraint solving calls is an importance metric because it is independent from the program implementation, the hardware configuration or different constraint solvers. The comparison results are shown in Tables IV and V. We do not show the number of tests, which is not affected by these optimizations.

TABLE IV. COMPARISON OF NUMBER OF CONSTRAINT SOLVING CALLS (2-WAY)

System	IPOG-C with Different Optimizations				
	Base	O1	O2	O3	All
Apache	15751	3903	12314	284	155
Bugzilla	2843	732	2352	57	50
GCC	34613	4753	31250	1032	631
SPIN-S	1183	478	1002	293	171
SPIN-V	10770	3679	9609	766	546
TCAS	828	597	535	59	42
C10	991	287	954	796	246

TABLE V. COMPARISON OF EXECUTION TIME (IN SECONDS)

System	IPOG-C with Different Optimizations				
	Base	O1	O2	O3	All
Apache	105.411	6.225	9.403	0.687	0.577
Bugzilla	2.808	0.904	1.763	0.328	0.296
GCC	683.599	24.462	59.429	1.809	1.139
SPIN-S	1.545	0.92	1.31	0.749	0.53
SPIN-V	81.323	18.239	11.169	1.124	0.889
TCAS	0.874	0.749	0.702	0.36	0.328
C10	1.014	0.53	0.89	0.828	0.515

The results in Tables IV and V suggest that the optimizations are very effective. Recall that the first optimization avoids validity check for all t-way target tuples in the beginning of test generation. This optimization is very effective when the system has a large number of t-way target tuples. For example, Apache contains 172 parameters and GCC has 199 parameters. They both have a large number of target tuples. With this optimization, the generation process runs 17 times faster for Apache, and 28 times faster for GCC.

The second optimization reduced the execution significantly, but not the number of constraint solving calls.. This is because this optimization is aimed to simplify the actual constraint solving process by only considering relevant constraints. Note that the number of constraint solving classes is also slightly decreased. This is because a parameter may not be involved in any constraint. In this case, no relevant constraints are found and thus no constraints need to be solved.

The third optimization seems to be the most effective optimization in the experiments. Recall that it records the solving history based on constraint groups to reduce

redundant solvings. This optimization is more effective with small constraint groups, where redundant solvings are more likely to happen. On the other hand, this optimization is less effective with large constraint groups. For example, for system C10, where 9 of 10 parameters belong to the same constraint group, this optimization is not very effective.

The three optimizations are complementary to each other and can be combined to further reduce the number of constraint solving calls and the execution time. In particular, for all of the real-life systems, the number of constraint solving calls was reduced by one or two orders of magnitude.

C. Evaluation of Different Factors

In this section, we explore how the performance of the entire test generation process is affected by different factors, including test strength, number of parameter, domain size and number of forbidden tuples. Each time we fix all the factors but one. We compare the test generation performance of our algorithm between with all the optimizations (i.e., the optimized version of algorithm IPOG-C) and without any optimization (the base version of algorithm IPOG-C).

1) Test Strength

We use system C1 to evaluate the performance of constrained test generation using different test strengths. Recall that C1 has 10 parameters of domain size 4. We record number of validity checks, number of times the constraint solver is called, and execution time in Table VI.

TABLE VI. TEST GENERATION WITH DIFFERENT TEST STRENGTHS

Test Strength	Num. of Target Tuples	Base		Optimized	
		Num. of Solving Calls	Time (sec)	Num. of Solving Calls	Time (sec)
2	683	644	0.67	77	0.31
3	7062	6869	3.9	121	0.32
4	47656	51787	64.17	122	0.40
5	218848	267421	1368.01	124	1.35
6	690816	Out of Memory	Out of Memory	124	14.39

One may find that as the test strength increases, the number of target tuples increase very fast. However, after those optimizations are applied, the number of constraint solving calls increases very slowly, and is even unchanged from strength 5 to 6. This is mainly due to the third optimization, which records the solving history for each constraint group. This system contains a single constraint group involving only 3 parameters. After all of the possible $4^3 = 64$ value combinations, have been checked, all validity checks can be handled by looking up the solving history. That is, no more solving calls are needed.

2) Number of Parameters

In this section, we evaluate the performance of the test generation process with respect to different numbers of parameters. We built 8 systems with the number of parameters ranging from 4, 6, 8 to 18. The constraint "(p1>p2 || p3>p2)" in system C1 is used for all of these 8 systems. The test strength is set to 3.

Fig. 6 show that as the number of parameters increases, the number of constraint solving calls and the execution time increase very fast for the base version, but very slow for the optimized version.

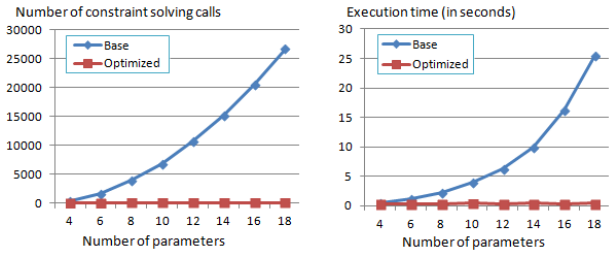

Figure 6. Performance w.r.t. different numbers of parameters

3) Domain Size

In this experiment, we still use system C1, but change the domain size to build 8 different systems. These systems have the same number of parameters and the same constraint, but the domain size increases from 2, 3, 4 until 9. The test strength is set to 3.

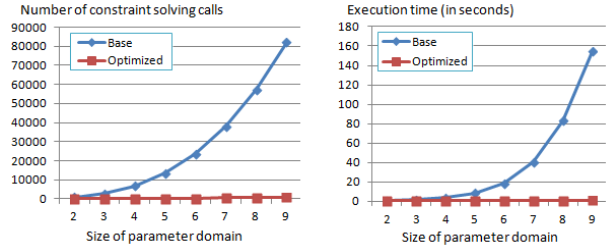

Figure 7. Performance w.r.t. different domain sizes

Again, Fig. 7 shows that as the domain size increases, the number of constraint solving calls and the execution time increase very fast for the base version, but very slow for the optimized version.

4) Number of Forbidden Tuples

We use all of the 10 synthesized systems in this section to evaluate how the performance of the test generation process changes when the number of forbidden tuples changes. As discussed earlier, these systems have the same parameter configuration but different constraints. Those constraints are carefully designed to control the number of forbidden tuples. The test strength is set to 3.

Fig. 8 shows that as the number of forbidden tuples increases, the number of constraint solving calls and the execution time increase very fast for the base version, but very slow for the optimized version.

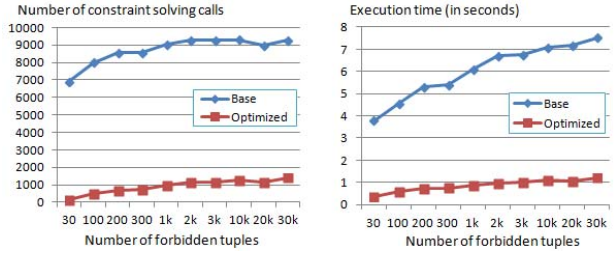

Figure 8. Performance w.r.t. different numbers of forbidden tuples

D. Comparison With Other Tools

In this section, we compare **ACTS** [1] (using the optimized IPOG-C algorithm) to other test generation tools. First we briefly introduce several existing test generation tools with constraint support.

CASA [7] integrates a SAT solver into a simulated annealing algorithm. Constraints are specified as Boolean formulas. We record the best result among five runs since this algorithm is not deterministic.

mAETG [6] integrates a SAT solver into an AETG-like algorithm. However, we did not make a comparison in this paper since mAETG is not available to public.

Ttuples [3] uses a greedy algorithm based on a property of (unconstrained) t-way test set, i.e., if two parameters have the same domain, it's safe to exchanging all their values. Constraints are specified as a set of forbidden tuples.

PICT [8] also adopts an AETG-like greedy algorithm. Constraints are specified in the form of logical expressions, but forbidden tuples are derived from the constraints to perform validity check.

These tools are compared in two dimensions: size of test set and execution time. However, it is important to note that size of test set mainly depends on the core test generation algorithms. Also even without constraints, the performances of different test generation algorithms are different. Furthermore, we did not make a comparison on the number of constraint solving calls or the number of validity checks, since we cannot obtain them from other tools.

Our comparison uses the 6 real-life systems and 10 synthesized systems introduced in Section VI.A. Since CASA and Ttuples cannot handle constraint expressions, we derived all forbidden tuples for 10 synthesized systems. The number of generated tests and the execution time for each system are shown in Table VII. The number of forbidden tuples is also listed. The test strength is set to 3. We limit the execution time within 500 seconds.

TABLE VII. COMPARISION OF DIFFERENT TOOLS (3-WAY)

System	Num. of Forbid. Tuples	CASA		Ttuples		PICT		ACTS	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)
Apache	7	-	-	-	-	202	176.01	173	**25.2**
Bugzilla	5	71	497.76	62	4.55	70	0.69	68	**0.61**
GCC	40	-	-	-	-	134	170.26	108	**35.52**
SPIN-S	13	103	187.51	127	0.30	113	**0.09**	98	1.82
SPIN-V	49	-	-	306	12.1	345	**4.92**	284	5.09
TCAS	3	405	99.7	402	0.27	409	**0.11**	405	0.55
C1	30	146	26.2	207	0.53	163	**0.06**	158	0.36
C2	100	164	47.23	202	1.31	171	**0.06**	168	0.58
C3	200	162	45.94	191	1.35	166	**0.07**	163	0.71
C4	300	157	59.15	200	2.69	166	**0.08**	161	0.74
C5	1000	157	72.86	196	5.03	170	**0.18**	160	0.87
C6	2000	161	68.83	195	7.12	163	0.96	161	**0.97**
C7	3000	166	70.14	188	14.94	162	1.09	160	**1.00**
C8	10000	160	99.12	196	257.75	163	17.34	164	**1.11**
C9	20000	150	131.5	-	-	162	242.1	157	**1.06**
C10	30000	155	114.63	-	-	161	461.5	158	**1.21**

* Dash (-) means the execution time is longer than 500 seconds.

We make several observations from Table VII.

CASA uses a non-deterministic generation algorithm. CASA generates relatively small test sets, but its execution time is relatively longer. This is because the simulated annealing algorithm is able to find a more optimal solution, but is usually much slower than greedy algorithms. Ttuples runs fast on small real life systems (SPIN-S, TCAS) and synthesized systems with small number of forbidden tuples. However, it takes much longer time to handle other systems. Generally speaking, it's more efficient to handle constraints using forbidden tuples when the number of forbidden tuples is small, but the test generation algorithm of Ttuples may not be efficient for large systems. For synthesized systems, the execution time of Ttuples increases very fast as the number of forbidden tuples increases. This is a disadvantage of using forbidden tuples. Similar to Ttuples, the execution time of PICT also increases very fast as the number of forbidden tuples increases. For other systems, PICT is fast and generates good results. ACTS generates relatively small test sets, and runs fast overall. For some systems, ACTS runs slightly slower than PICT, but the differences are very small. ACTS runs much faster than other tools on systems with large number of parameters (Apache, GCC) and systems with large number of forbidden tuples (C9, C10), exhibiting very good scalability. This can be a significant advantage when we deal with systems with more complex constraints.

Fig. 9 shows how execution time changes as number of forbidden tuples in 10 synthesized systems increases. When the number of forbidden tuples is small, Ttuples, PICT and ACTS have similar execution time, while CASA takes longer time to finish. However, as the number of forbidden tuples increases, the execution time of Ttuples and PICT increases significantly. In contrast, the execution time of CASA increases slowly, and the execution time for ACTS remains almost unchanged. The reason is that CASA and ACTS use constraint solver for validity checks, and do not have to maintain a large number of forbidden tuples during test generation. This demonstrates a major advantage of using a constraint solver instead of forbidden tuples.

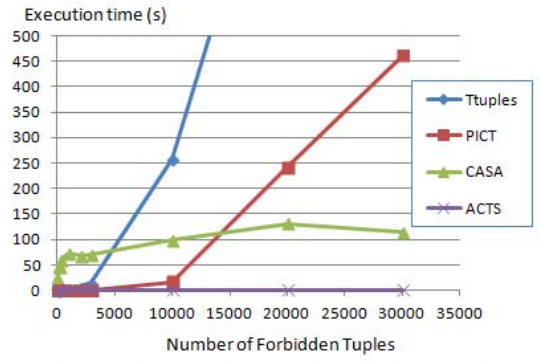

Figure 9. Camparison of Different Number of Tuples

E. Threats to Validity

The main external threat to validity is that the benchmark systems used in our experiments may not be representative of real-life applications. We used both real-life and synthesized systems to reduce this threat. The

internal threat to validity is mainly due to potential mistakes made in the experiments. We tried to automate the experiments as much as possible to reduce chances of error. For example, a tool was used to automatically derive forbidden tuples from the constraint expressions of the synthesized systems. Bugs may also exist in the implementation of the test generation algorithm. To reduce this threat, we used an independent procedure to check the validity of each test and the coverage.

VI. Related Work

We focus our discussion on work that handles constraints using a constraint solver. Garvin et al. integrated a SAT solver into a meta-heuristic search algorithm, called simulated annealing, for constrained combinatorial test generation [9][10]. It was found that integration with the original version of the search algorithm did not produce competitive results, both in terms of number of tests and test generation time. Thus, a couple of changes were made to the original search algorithm to improve the results. The modified search algorithm could produce a different test set than the original search algorithm. This is in contrast to our work, where our optimizations do not change the original test generation algorithm, i.e., IPOG. In particular, our optimizations reduce the execution time spent on constraint handling, but do not change the size of the test set.

Cohen et al. integrated a SAT solver into an AETG-like test generation algorithm [6] [7]. They also proposed two optimizations to improve the overall performance. In their optimizations, the history of the SAT solver is exploited to reduce the search space of the original test generation algorithm. Like the work in [9], their optimizations require changes to the original test generation algorithm, and thus could produce a different test set. In addition, their optimizations require access to the solving history and are thus tightly coupled with the SAT solver. This is in contrast with our optimizations, which do not change the original test generation algorithm and are independent from the constraint solver.

Recent work has applied combinatorial testing to software product lines. A software product line is a family of products that can be created by combining a set of common features. In this domain, constraint handling is a must because dependencies naturally exist between different features. Hervieu et al [12] developed a constraint programming approach for pairwise testing of software product lines. The focus of their work is on the conversion of the pairwise test generation problem to a constraint-programming problem. In particular, they formulated a global constraint to achieve pairwise coverage. Their work relies on the underlying constraint solver to achieve the best result. That is, they do not explicitly address the optimization problem.

Perrouin et al. [19] addressed the scalability of a constraint solver in the context of t-way testing of software product lines. Specifically, they address the problem that current constraint solvers have a limit in the number of clauses they can solve at once. They use a divide-and-conquer strategy to divide the t-way test generation problem for the entire feature model into several sub-problems. Their work addresses a different problem than, and is complementary to, our work, which tries to reduce the number of calls to a constraint solver and to remove constraints that are not relevant in a constraint solving call.

Johansen et al. [13] developed an algorithm called ICPL that applies t-way testing to software product lines. Similar to our algorithm, algorithm ICPL includes several optimizations to reduce the number of calls to a constraint solver. There are two major ideas in their optimizations that are closely related to our optimizations. The first idea is to postpone removal of invalid target combinations (called t-sets in [13]). This achieves an effect similar to our first optimization, i.e., avoiding unnecessary validity checks of target combinations. However, there are two important differences. First, algorithm ICPL uses a heuristic to determine at which point to remove invalid target combinations. In contrast, our algorithm, IPOG-C, removes invalid target combinations during vertical growth, without using any heuristic condition. Second, they have very different motivations. Algorithm ICPL adopts a target combination-oriented framework, where the main loop iterates through the set of target combinations and covers them as they are encountered. Removing invalid combinations up front would cause two constraint solving calls for many valid combinations. (The other call is needed when a valid combination is actually covered in a test.) In contrast, our algorithm largely uses a test-oriented framework, where we try to determine each value in a test such that it covers as many combinations as possible. The key insight in our optimization is that if a test is found valid, then all the combinations covered by this test would be valid, and thus do not have to be explicitly checked.

The second optimization idea in algorithm ICPL that is closely related to ours is trying to check the validity of a t-way combination without actually calling the constraint solver. Algorithm ICPL is recursive in which a t-way test set is built by extending a (t-1)-way test set. The set of invalid combinations is maintained at each strength in the recursive process. An invalid t-way target combination is identified if it is an extension of an invalid (t-1)-way combination. In contrast, our algorithm records the solving history, which is used to determine the validity of a target combination as well as a test without calling the constraint solver. Also, our algorithm is not recursive, and does not maintain a set of invalid target combinations.

It is important to note that work on testing software product lines assumes Boolean parameters and constraints in the form of Boolean logic expressions. In contrast, our work does not have this restriction. Furthermore, software product lines typically have a large number of constraints but a small t-way test set. As a result, some optimizations that are effective for software product lines may not be very effective for general systems, and vice versa.

Finally we note that many optimization techniques are employed inside existing constraint solvers. In principle, our second and third optimizations are similar to constraint slicing and caching strategies used in some constraint solvers like zChaff [18] and STP [2]. These optimizations are also used outside a constraint solver in program analysis tools such as EXE [2]. However, we differ in that our

optimizations work together with the combinatorial test generation algorithm and leverage its incremental framework to achieve maximal performance improvements. To our best knowledge, this is the first time these techniques are applied in a way that is integrated with the combinatorial test generation framework.

VII. Conclusion and Future Work

In this paper, we present an efficient algorithm, called IPOG-C, for constrained combinatorial test generation. The major contribution of our work is three optimizations employed by algorithm IPOG-C to improve the performance of constraint handling. These optimizations try to reduce the number of calls to a constraint solver. When such a call cannot be avoided, these optimizations try to reduce the number of constraints that have to be solved. We show that these optimizations can be applied to other test generation algorithms. Experiment results show that these optimizations can achieve performance improvements of up to two orders of magnitude. The IPOG-C algorithm is implemented in a combinatorial test generation tool, i.e., named ACTS, which is freely available to public. A comparative evaluation suggests that ACTS can perform significantly better than other tools for systems that have more complex constraints.

There are several directions to continue our work. First, we want to conduct more experiments to evaluate the effectiveness of our algorithm. In particular, the real-life systems in our experiments have a very small number of forbidden tuples. We want to investigate whether this is the case in general and if possible, apply our algorithm to real-life systems with a large number of forbidden tuples. Second, we want to develop efficient schemes to parallelize our algorithm. For example, we could divide the complete set of target combinations into several subsets, and then assign these subsets to different cores or processors. As another example, when we try to select the best value of a parameter, we could employ multiple cores or processors to determine the weight of each value. Finally, we plan to investigate how to integrate our algorithm into an existing test infrastructure. Most work on combinatorial testing only addresses the test generation problem. Combinatorial testing can generate a large number of tests. It is thus particularly important to streamline the entire test process, i.e., integrate our test generation tool with other tools that automate test execution and test evaluation.

Acknowledgment

This work is partly supported by three grants (70NANB9H9178, 70NANB10H168, 70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST) and a grant (61070013) of National Natural Science Foundation of China.

DISCLAIMER: NIST does not endorse or recommend any commercial product neither referenced in this paper nor imply that the referenced product is necessarily the best.

References

[1] ACTS, http:// csrc.nist.gov/acts/

[2] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, Dawson Engler, "EXE: Automatically Generating Inputs of Death," ACM Transactions on Information and System Security (TISSEC) Volume 12, No. 2, December 2008

[3] A. Calvagna, A. Gargantini, "T-wise combinatorial interaction test suitees construction based on coverage inheritance," In: Software Testing, Verification and Reliability, 2009

[4] Choco Solver, http://www.emn.fr/z-info/choco-solver/

[5] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," IEEE Transactions On Software Engineering, 23(7):437–444, 1997

[6] M.B. Cohen, M.B. Dwyer, J. Shi., "Interaction testing of highly-configurable systems in the presence of constraints," In: 5th international symposium on software testing and analysis, pp 129–139, 2007

[7] M.B. Cohen, M.B. Dwyer, J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach", IEEE Trans. Softw. Eng. 34, 633–650, 2008

[8] J. Czerwonka, "Pairwise testing in real world," In: 10th Pacific northwest software quality conference, pp 419–430, 2006

[9] B.J. Garvin, M.B. Cohen, and M.B. Dwyer. "An improved meta-heuristic search for constrained interaction testing," In: 1st international symposium on search based software engineering, pp 13–22, 2009

[10] B.J. Garvin, M.B. Cohen, and M.B. Dwyer, "Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing," Empirical Software Engineering (EMSE), 16(1), pp.61-102, 2011

[11] Mats Grindal, Jeff Offutt, Jonas Mellin, "Managing conflicts when using combination strategies to test software," Proceedings of the 2007 Australian Software Engineering Conference (ASWEC 2007), pp. 255–264, 2007

[12] Aymeric Hervieu, Benoit Baudry, Arnaud Gottlieb, "PACOGEN : Automatic Generation of Test Configurations from Feature Models," Proc. of Int. Symp. on Soft. Reliability Engineering (ISSRE'11), 2011

[13] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey, "An Algorithm for Generating T-wise Covering Arrays from Large Feature Models," In Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC 2012), pages 46-55.

[14] D.R. Kuhn, M.J. Reilly, "An investigation of the applicability of design of experiments to software testing," Proceedings of 27th NASA/IEEE Software Engineering Workshop, 2002; 91–95

[15] D.R. Kuhn, D.R. Wallace, Jr. A.M. Gallo, "Software fault interactions and implications for software testing," Software Engineering, IEEE Transactions on, 2004

[16] D.R. Kuhn, Vadim Okum, "Pseudo-exhaustive testing for software," In SEW '06: IEEE/NASA Software Engineering Workshop, Los Alamitos, CA, USA, 2006

[17] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG: A general strategy for t-way software testing," In: 14th international conference on the engineering of computer-based systems, 2007

[18] Y. S. Mahajan and S. M. Z. Fu, "Zchaff2004: An efficient sat solver," in SAT 2004, pp. 360–375, 2004

[19] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, Yves le Traon, "Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines," 2010 IEEE International Conference on Software Testing Validation and Verification, 2010

[20] D.R. Wallace, D.R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," International Journal of Reliability, Quality and Safety Engineering 2001