LETTER *Special Section on Software Reliability Engineering*

Using Satisfiability Solving for Pairwise Testing in the Presence of Constraints

Toru NANBA[†*], *Nonmember*, **Tatsuhiro TSUCHIYA**[†a)], *Member*, *and* **Tohru KIKUNO**[†**], *Fellow*

SUMMARY This letter discusses the applicability of boolean satisfiability (SAT) solving to pairwise testing in practice. Due to its recent rapid advance, using SAT solving seems a promising approach for search-based testing and indeed has already been practiced in test generation for pairwise testing. The previous approaches use SAT solving either for finding a small test set in the absence of parameter constraints or handling constraints, but not for both. This letter proposes an approach that uses a SAT solver for constructing a test set for pairwise testing in the presence of parameter constraints. This allows us to make full use of SAT solving for pairwise testing in practice.

key words: *SAT, software testing, pairwise testing, test set generation, constraints*

1. Introduction

This letter proposes an approach that uses boolean satisfiability (SAT) solving for finding a test set for *pairwise testing*. Pairwise testing or all-pairs testing is a well-practiced software testing method [1]–[3]. The idea of this testing method is to test at least once every possible value pair for every pair of the test parameters. We call this coverage criterion simply *pairwise coverage*. The rationale behind pairwise testing stems from the observation that most faults are caused by interaction of at most two parameters [4] and the fact that test sets that meet pairwise coverage can be much smaller than those for exhaustive testing.

As an example, consider the testing model shown in Table 1. This model, which is taken from the work by Cohen, Dwyer and Shi [5], models the testing of mobile phone products. It specifies five parameters and the values they may take. A test case is a vector consisting of five parameter values, one for each parameter. Table 2 shows an example of a test set that meets the pairwise coverage. One can see that for any two of the parameters, every possible pair of values for the parameters occurs in at least one of the test cases.

In the above example, we implicitly assume that any value of any parameter can occur with any value of any other parameter in a test case. However, in practice it is usually the case that some particular combinations of parameter values cannot occur simultaneously. For example, consider the following constraint.

Table 1 A testing model.

Parameter	Possible Values
Display	16MC, 8MC, BW
Email Viewer	Graphical, Text, None
Camera	2MP, 1MP, None
Video Camera	Yes, No
Video Ringtones	Yes, No

MC: Million Colors, BW: Black and White, MP: Megapixel

Table 2 A test set for pairwise testing ignoring constraints.

	Display	Email Viewer	Camera	Video Camera	Video Ringtones
1	BW	Text	1MP	Yes	Yes
2	16MC	Graphical	2MP	Yes	Yes
3	8MC	Graphical	1MP	No	No
4	16MC	Text	None	No	No
5	8MC	None	None	Yes	Yes
6	BW	None	2MP	No	No
7	BW	Graphical	None	No	Yes
8	16MC	None	1MP	Yes	No
9	8MC	Text	2MP	Yes	No

- Video camera requires a camera and a color display.

This constraint yields the following forbidden pairs, which cannot occur in any test case.

(Camera, Video Camera) = (None, Yes)
(Display, Video Camera) = (BW, Yes)

Note that test cases 1 and 5 in Table 2 do not satisfy this constraint.

Thus, in practice, test set generation for pairwise testing needs to address the following two issues:

- to find a test set that consists of a small number of tests, and
- to find a test set such that every test case satisfies the constraints on parameter values.

There is a substantial body of work on test set construction for pairwise testing. However, many of the previously known constructions can only deal with such constraints in an ad hoc manner or even totally ignore the problem of handling constraints. In spite of the apparent simplicity, this problem is subtle enough to require sophisticated treatment, as explained in the next section in detail.

To address the above two issues, this letter proposes the use of boolean satisfiability (SAT) solving. SAT is the problem of determining whether or not there is a value assignment that causes a given boolean formula to evaluate to true. It is well-known that SAT is NP-complete; thus it is

Manuscript received December 25, 2011.
Manuscript revised April 6, 2012.
[†]The authors are with Osaka University, Suita-shi, 565-0871 Japan.
[*]Presently, with the Chugoku Electric Power Co., Inc.
[**]Presently, with Osaka Gakuin University.
a) E-mail: t-tutiya@ist.osaka-u.ac.jp
DOI: 10.1587/transfun.E95.A.1501

strongly conjectured that no efficient algorithm exists. In practice, however, SAT solvers — solvers to the problem — see a drastic advance in recent years. They work very fast for many problem instances that arise from practical problems; thus reducing a particular problem to SAT and then using a SAT solver has now become a standard approach to search problems.

In previous studies, the use of SAT solving for test generation for pairwise testing has been discussed for two purposes; namely, finding a small test set in the absence of parameter constraints and handling constraints on parameter values.

In [6], [7], SAT solving is used to find small *covering arrays*. A covering array is a term in mathematics which is basically equivalent to a test set for pairwise testing. Research in this line places emphasis on finding minimal covering arrays and does not consider constraints that occur in practical testing.

On the other hand, the authors of [5] propose to use a SAT solver to handle constraints. Their approach is tied with the greedy test set construction approach, where a test set is incrementally generated by adding one test case at one time. Basically SAT solving is only used for checking if a candidate of a test case satisfies the constraints. In [8] SAT solving is used the same way in simulated annealing-based test set construction.

This article is a revised version of our earlier workshop paper [9]. Here we include some concrete examples to demonstrate how the proposed approach works and elaborate the approach more fully, especially the first step of the two-step approach, which was only briefly described in [9].

2. Pairwise Testing in the Presence of Constraints

As discussed in the first section, it is necessary to consider constraints on parameter values in pairwise testing in practice. Thus the problem of generating a test set for pairwise testing can be considered a combinatorial optimization problem where the objective function is the number of test cases which should be minimized while satisfying the following coverage criterion and constraints.

- Pairwise coverage: For any pair of parameters, all *possible* pairs of values for the parameters must occur in at least one test case.
- Constraints on parameter values: Any test case must satisfy a given set of constraints.

Here by *possible* pairs of parameter values, we mean pairs of parameter values that can occur in at least one test case that satisfies all the constraints. We call a pair that is not possible a *forbidden* pair. We call a test case *feasible* if it satisfies all the constraints; it is *infeasible* otherwise. As explained in the next section, it is necessary to find all forbidden pairs of parameter values in encoding the test set generation problem into SAT.

The difficulty in dealing with constraints stems from the fact that a given set of "explicit" constraints may imply "hidden" constraints. Consider two constraints as follows:

- Video camera requires a camera and a color display.
- Video ringtones cannot occur with No video camera.

From these constraints, it is straightforward to see that the following pairs cannot occur in any test case.

(Camera, Video Camera) = (None, Yes)
(Display, Video Camera) = (BW, Yes)
(Video Camera, Video Ringtones) = (No, Yes)

However it is not easy to notice that these forbidden pairs imply the following hidden forbidden pairs:

(Camera, Video Ringtones) = (None, Yes)
(Display, Video Ringtones) = (BW, Yes)

Finding all such forbidden pairs needs a systematic handling of constraints. In our approach a SAT solver is used for this purpose.

3. Proposed Approach

The whole process of constructing a test set consists of two steps. The first step discovers all forbidden value pairs. These value pairs are necessary in the second step in which a test set is created. The second step produces a set of test cases such that all possible value pairs are covered.

3.1 Finding All Forbidden Pairs

It would be possible to check all pairs of parameter values using SAT solving; but this approach would be time-consuming because the number of such pairs is quadratic both in the number of parameters and the number of values for a parameter.

We use a quicker method as follows: First we construct a test set for pairwise testing without considering constraints. There are many algorithms that can efficiently construct a test set for pairwise testing when there is no constraint on parameter values [10]–[12]. Hence we could use any of these algorithms in this step.

Then for each test case in the test set, we check if that test case contains any forbidden pairs. This can be done by invoking a single SAT instance, as explained below. If it turns out that the test case violates some of the constraints — meaning that it may contain one or more forbidden pairs — then we will check each of the $n*(n-1)$ value pairs in the test case using SAT solving, where n is the number of parameters. If the test case is feasible, every value pair that occurs in the test case is possible. Note that by doing this, all pairs can be tested and that the pairs that are actually checked are limited to those contained in infeasible test cases.

Now we show how SAT solving can be used to determine if a test case is infeasible, i.e., may contain forbidden pairs. We use boolean variables $x_{j,k}$ for $1 \leq j \leq n$ and $1 \leq k \leq P(j)$, where n is the number of parameters and $P(j)$ is the number of possible values for the jth parameter. We denote the kth value of the jth parameter by integer k ($1 \leq k \leq P(j)$). The variable $x_{j,k}$ being true means that the jth parameter has value k, i.e., its kth value. Obviously a

given test case $(k_1, \cdots, k_n)$ can be represented as follows.

$$Test := \bigwedge_{\forall j:1\le j\le n} \left(x_{j,k_j} \wedge \bigwedge_{\forall k:1\le k\le P(j), k\neq k_j} \neg x_{j,k} \right)$$

where the symbol := is read "is defined to equal". For example, suppose that we use the test set shown in Table 2 to find all forbidden pairs and that now we want to check whether the first test case is feasible or infeasible. The first test case is represented by integer vector (3, 2, 2, 1, 1) and thus we have:

$$\begin{aligned} &Test := \\ &\neg x_{1,1} \neg x_{1,2} x_{1,3} \neg x_{2,1} x_{2,2} \neg x_{2,3} \neg x_{3,1} x_{3,2} \neg x_{3,3} x_{4,1} \neg x_{4,2} \\ &x_{5,1} \neg x_{5,2} \end{aligned}$$

It is straightforward to see that the constraints on the parameter values can be represented as a boolean formula over these boolean variables. Let *Cons* denote this boolean formula. An important observation here is that *Cons* can be derived from only the given, explicit constraints. For example, consider the two constrains discussed in Sect. 2. In this case we have:

$$\begin{aligned} &Cons := \\ &\quad (x_{4,1} \rightarrow ((x_{3,1} \vee x_{3,2}) \wedge (x_{1,1} \vee x_{1,2})) \\ &\wedge (x_{5,1} \rightarrow \neg x_{4,2}) \end{aligned}$$

where $\rightarrow$ means implication.

Now whether the test case is feasible, i.e., meets the constraints can be determined by checking the satisfiability of:

$$Test \wedge Cons$$

If it is satisfiable, then the test case is feasible; otherwise the test case is infeasible. For the first test case in Table 2 this formula is unsatisfiable; thus one can conclude that it may contain some forbidden pairs.

If the test case is found to be infeasible, we check for every pair that occurs in the test case whether it is a forbidden pair or not. This check is performed with a SAT solver as follows. Suppose that the pair to be checked is value k_1 at the j_1th parameter and value k_2 at the j_2th parameter. We denote this pair as $((j_1, k_1), (j_2, k_2))$. Again we use boolean variables $x_{j,k}$ for $1 \le j \le n$ and $1 \le k \le P(j)$ to represent that the jth parameter has the kth value. Any test case that contains the pair is represented as the conjunction of the following formulas:

$$\begin{aligned} &AtLeast := \bigwedge_{\forall j:1\le j\le n} \bigvee_{\forall k:1\le k\le P(j)} x_{j,k} \\ &AtMost := \bigwedge_{\forall j:1\le j\le n} \bigvee_{\forall k,k':1\le k<k'\le n} (\neg x_{j,k} \vee \neg x_{j,k'}) \\ &Pair_{((j_1,k_1),(j_2,k_2))} := x_{j_1,k_1} x_{j_2,k_2} \end{aligned}$$

Formula *AtLeast* specifies that any parameter has some value. Formula *AtMostT* specifies that any parameter has at most one value. Formula *Pair* specifies the pair $((j_1, k_1), (j_2, k_2))$. It is not difficult to see that value assignments that satisfy all the three formulas have one-to-one correspondence to test cases that contain the given pair. Accordingly, test cases that contain the given pair *and* satisfy the constraints have one-to-one correspondence to satisfying value assignments for:

$$AtLeast \wedge AtMost \wedge Pair \wedge Cons$$

The pair is a forbidden pair if and only if the formula is unsatisfiable.

For example, consider the pair BW at Display and Yes at Video Ringtones (i.e., ((1, 3), (5, 1)). Whether it is forbidden or not is checked by the satisfiability check of:

$$\begin{aligned} &((x_{1,1} \vee x_{1,2} \vee x_{1,3})(x_{2,1} \vee x_{2,2} \vee x_{2,3}) \\ &\ (x_{3,1} \vee x_{3,2} \vee x_{3,3})(x_{4,1} \vee x_{4,2})(x_{5,1} \vee x_{5,2})) \\ &\wedge ((\neg x_{1,1} \vee \neg x_{1,2})(\neg x_{1,1} \vee \neg x_{1,3})(\neg x_{1,2} \vee \neg x_{1,3}) \\ &\quad (\neg x_{2,1} \vee \neg x_{2,2})(\neg x_{2,1} \vee \neg x_{2,3})(\neg x_{2,2} \vee \neg x_{2,3}) \\ &\quad (\neg x_{3,1} \vee \neg x_{3,2})(\neg x_{3,1} \vee \neg x_{3,3})(\neg x_{3,2} \vee \neg x_{3,3}) \\ &\quad (\neg x_{4,1} \vee \neg x_{4,2})(\neg x_{5,1} \vee \neg x_{5,2})) \\ &\wedge x_{1,3} x_{5,1} \\ &\wedge ((x_{4,1} \rightarrow ((x_{3,1} \vee x_{3,2}) \wedge (x_{1,1} \vee x_{1,2}))) \\ &\quad \wedge (x_{5,1} \rightarrow \neg x_{4,2})) \end{aligned}$$

This formula is unsatisfiable; thus one can conclude that the pair ((1, 3), (5, 1)) is forbidden.

Among the nine test cases of Table 2, only #1, #5 and #7 are infeasible with respect the two constraints described in Sect. 2; thus all forbidden pairs can be found by checking only the pairs that occur in the three test cases. There are a total of 27 different pairs in the three infeasible test cases. Note that this number is much smaller than the number of all pairs, which is 67 $(= 3 * (3 + 3 + 2 + 2) + 3 * (3 + 2 + 2) + 3 * (2 + 2) + 2 * 2)$. As a result, one can obtain a total of five forbidden pairs, ((1, 3), (4, 1)), ((1, 3), (5, 1)), ((3, 3), (4, 1)), ((3, 3), (5, 1)), and ((4, 2), (5, 1)), which are identical to those described in Sect. 2.

3.2 Searching for a Small Test Set

Once all forbidden pairs of parameter values have been discovered, the next step is to construct a test set that covers all parameter value pairs except those forbidden pairs. This is done by iteratively solving the satisfiability problem.

3.2.1 SAT Encoding

As SAT is a decision problem rather than an optimization problem, we obtain a final test set by using a SAT solver to solve a series of the decision problem as follows:

Given a size l, is there a test set of that size?

Here we show the encoding of this problem into SAT.

We use a set of boolean variables $x_{i,j,k}$, $1 \le i \le l, 1 \le j \le n, 1 \le k \le P(j)$. The variable $x_{i,j,k}$ being true means that the jth parameter of the ith test case has the kth value.

The boolean formula to be checked must be encoded

$$\left(\begin{array}{l}((x_{1,1,1} \vee x_{1,1,2} \vee x_{1,1,3})(x_{1,2,1} \vee x_{1,2,2} \vee x_{1,2,3}) \\ (x_{1,3,1} \vee x_{1,3,2} \vee x_{1,3,3})(x_{1,4,1} \vee x_{1,4,2})(x_{1,5,1} \vee x_{1,5,2})) \\ \wedge ((\neg x_{1,1,1} \vee \neg x_{1,1,2})(\neg x_{1,1,1} \vee \neg x_{1,1,3})(\neg x_{1,1,2} \vee \neg x_{1,1,3}) \\ \quad (\neg x_{1,2,1} \vee \neg x_{1,2,2})(\neg x_{1,2,1} \vee \neg x_{1,2,3})(\neg x_{1,2,2} \vee \neg x_{1,2,3}) \\ \quad (\neg x_{1,3,1} \vee \neg x_{1,3,2})(\neg x_{1,3,1} \vee \neg x_{1,3,3})(\neg x_{1,3,2} \vee \neg x_{1,3,3}) \\ \quad (\neg x_{1,4,1} \vee \neg x_{1,4,2})(\neg x_{1,5,1} \vee \neg x_{1,5,2})) \\ \wedge ((x_{1,4,1} \rightarrow ((x_{1,3,1} \vee x_{1,3,2}) \wedge (x_{1,1,1} \vee x_{1,1,2}))) \\ \quad \wedge (x_{1,5,1} \rightarrow \neg x_{1,4,2}))\end{array}\right) \wedge \ldots \wedge \left(\begin{array}{l}((x_{l,1,1} \vee x_{l,1,2} \vee x_{l,1,3})(x_{l,2,1} \vee x_{l,2,2} \vee x_{l,2,3}) \\ (x_{l,3,1} \vee x_{l,3,2} \vee x_{l,3,3})(x_{l,4,1} \vee x_{l,4,2})(x_{l,5,1} \vee x_{l,5,2})) \\ \wedge ((\neg x_{l,1,1} \vee \neg x_{l,1,2})(\neg x_{l,1,1} \vee \neg x_{l,1,3})(\neg x_{l,1,2} \vee \neg x_{l,1,3}) \\ \quad (\neg x_{l,2,1} \vee \neg x_{l,2,2})(\neg x_{l,2,1} \vee \neg x_{l,2,3})(\neg x_{l,2,2} \vee \neg x_{l,2,3}) \\ \quad (\neg x_{l,3,1} \vee \neg x_{l,3,2})(\neg x_{l,3,1} \vee \neg x_{l,3,3})(\neg x_{l,3,2} \vee \neg x_{l,3,3}) \\ \quad (\neg x_{l,4,1} \vee \neg x_{l,4,2})(\neg x_{l,5,1} \vee \neg x_{l,5,2})) \\ \wedge ((x_{l,4,1} \rightarrow ((x_{l,3,1} \vee x_{l,3,2}) \wedge (x_{l,1,1} \vee x_{l,1,2}))) \\ \quad \wedge (x_{l,5,1} \rightarrow \neg x_{l,4,2}))\end{array}\right)$$

$$\wedge (x_{1,1,1}x_{1,2,1} \vee \ldots \vee x_{l,1,1}x_{l,2,1}) \wedge (x_{1,1,1}x_{1,2,2} \vee \ldots \vee x_{l,1,1}x_{l,2,2}) \wedge \ldots \wedge (x_{1,4,2}x_{1,5,2} \vee \ldots \vee x_{l,4,2}x_{l,5,2})$$

Fig. 1 Encoded boolean formula.

Table 3 A test set that satisfies pairwise coverage and constraints.

	Display	Email Viewer	Camera	Video Camera	Video Ringtones
1	8MC	None	2MP	Yes	No
2	8MC	Graphical	1MP	Yes	Yes
3	16MC	Text	2MP	Yes	Yes
4	16MC	None	1MP	Yes	Yes
5	16MC	Graphical	None	No	No
6	BW	Graphical	2MP	No	No
7	BW	None	None	No	No
8	BW	Text	1MP	No	No
9	8MC	Text	None	No	No

such that a satisfying value assignment to the variables exists if and only if a test set of size l exists. The encoding presented here enforces a more strong condition: satisfying value assignments have one-to-one correspondence to test sets that satisfy pairwise coverage and constraints. The boolean formula is the conjunction of the following formulas:

$$\bigwedge_{\forall i: 1 \le i \le l} (AtMost \wedge AtLeast \wedge Cons)\langle x_{i,j,k}/x_{j,k}\rangle$$

$$\bigwedge_{\forall((j_1,k_1),(j_2,k_2)) \in PossiblePairs} \bigvee_{\forall i: 1 \le i \le l} Pair_{(j_1,k_1),(j_2,k_2)}\langle x_{i,j,k}/x_{j,k}\rangle$$

where $formula\langle x_{i,j,k}/x_{j,k}\rangle$ represents $formula$ with $x_{i,j,k}$ substituted for each $x_{j,k}$ and $PossiblePairs$ is a set of all possible parameter value pairs.

The first formula specifies that the satisfying value assignment represents a test set of size l such that any of the l test cases satisfies the constraints. The second formula specifies that every possible pair must occur in at least one of the l test cases. Figure 1 shows the whole formula for the running example. The test set shown in Table 3 is obtained from a satisfying value assignment yielded by a SAT solver when $l = 9$. Obviously this test set is optimal in size.

3.2.2 Binary Search

Now we have a procedure that can determine the existence of a test case of a given size. Running this procedure with varying size, we can obtain the minimum test set, unless the SAT solver cannot complete computation within a reasonable time. It is often the case that if the given size becomes close to the optimum, even a recent SAT solver cannot terminate its computation within a reasonable time. Therefore we set a timeout period for each invocation of a SAT solver and if the solver cannot finish the job within the timeout period, the problem instance is treated as if it were unsatisfiable. To vary the size, binary search is used as in [13]. This allows to avoid checking of all possible sizes.

4. Experimental Results

We wrote a C program that implements the proposed approach. This program repeatedly runs MiniSAT [14], one of the best known modern SAT solvers. In the step of finding forbidden pairs, we use an algebraic method proposed in [15] to construct a test set that meets pairwise coverage.

We created a total of 16 artificial problem instances. For each problem instance, we randomly created several explicit forbidden value pairs as constraints.

We applied, to each of the instances, the proposed approach with two different timeout times *to*: 60 seconds and 300 seconds. The machine used in the experiments is a Linux machine equipped with a 2 GHz Intel Core 2 Duo and 1 Gbyte memory. The execution time of our program was typically a few times of the time out period.

For comparison purposes, we also used PICT, a test set generator developed at Microsoft [16]. PICT uses the greedy approach and can support constraint handling. PICT runs much faster than ours, whereas it does not aim to find the minimum test set. The execution time of PICT was typically around a second.

Table 4 summarizes the problem instances and the size of the obtained test sets. The entries in the Model column specify the number of parameters and the number of possible values for them. For example, the first entry $2^{20}5^5$ states that there are 20 parameters that have two possible values and five parameters that have five values.

As can be seen in the table, the SAT-based approach yielded smaller test sets than PICT for many of the problem instances. The few exceptions (#3, #8, #15) occurred when a SAT problem could not be solved within the time out period.

5. Conclusion

In this letter, we described a test set construction for pairwise testing. Our approach uses SAT solving to deal with constraints on parameters and to obtain small test sets. This idea allows us to make full use of modern fast SAT solvers. Experimental results showed that our approach usually yields smaller test sets than does a greedy approach. However this comes with cost of long computation time. A

Table 4 Results.

	Model	No. given forbidden pairs	SAT *to*=60s	SAT *to*=300s	PICT
1	$2^{20}5^{5}$	13	33	**27**	42
2	$2^{50}3^{30}$	40	**20**	**20**	37
3	$3^{20}6^{15}$	18	87	86	**80**
4	$4^{10}7^{3}$	7	50	**50**	63
5	$2^{20}5^{10}8^{5}$	18	105	**99**	103
6	$3^{15}5^{5}7^{3}$	12	**55**	**55**	67
7	$2^{30}3^{20}4^{10}$	30	**27**	**27**	40
8	$7^{10}8^{10}9^{10}$	15	—	—	**175**
9	$2^{5}3^{4}4^{3}5^{2}$	7	**25**	**25**	32
10	$2^{25}4^{10}7^{2}8^{1}$	19	**56**	**56**	72
11	$3^{15}4^{10}7^{5}8^{2}$	16	95	**92**	95
12	$4^{10}5^{5}8^{2}9^{1}$	9	**75**	**75**	89
13	$3^{5}4^{3}6^{2}8^{1}9^{1}$	6	**72**	**72**	75
14	$2^{30}3^{20}4^{10}5^{8}6^{3}$	34	**49**	**49**	63
15	$2^{10}3^{10}4^{10}5^{8}9^{5}$	22	140	137	**127**
16	$2^{20}4^{15}5^{4}7^{3}8^{2}$	22	80	**77**	92

possible future direction of the work is to shorten computing time by, for example, improving SAT encoding. Indeed, although no care is taken for parameter constraints, encodings in [6] and [7] are more involved to reduce SAT solving time than the one presented here. Adapting our constraint handling to these SAT encodings may worth studying.

Acknowledgments

Tatsuhiro Tsuchiya thanks Dr. Takashi Kitamura and Dr. Do Thi Bich Ngoc at The National Institute of Advanced Industrial Science and Technology (AIST) for discussion on an earlier draft of this article. This work was supported in part by Grants-in-Aid for Scientific Research, MEXT/JSPS (No. 23500046).

References

[1] R. Mandl, "Orthogonal latin squares: An application of experiment design to compiler testing," Commun. ACM, vol.28, no.10, pp.1054–1058, 1985.

[2] K. Tatsumi, "Test case design support system," Proc. International Conference on Quality Control (ICQC'87), pp.615–620, 1987.

[3] 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 Trans. Softw. Eng., vol.23, no.7, pp.437–444, 1997.

[4] D.R. Kuhn, D.R. Wallace, and A.M. Gallo, Jr., "Software fault interactions and implications for software testing," IEEE Trans. Softw. Eng., vol.30, no.6, pp.418–421, 2004.

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

[6] B. Hnich, S. Prestwich, and E. Selensky, "Constraint-based approaches to the covering test problem," Recent Advances in Constraints, ed. B. Faltings, A. Petcu, F. Fages, and F. Rossi, Lect. Notes Comput. Sci., vol.3419, pp.172–186, Springer Berlin/Heidelberg, 2005.

[7] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers," Proc. 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'10, pp.112–126, Springer-Verlag, Berlin, Heidelberg, 2010.

[8] B. Garvin, M. Cohen, and M. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," Empirical Software Engineering, vol.16, no.1, pp.61–102, 2011.

[9] T. Nanba, T. Tsuchiya, and T. Kikuno, "Constructing test sets for pair-wise testing: A SAT-based approach," Proc. 2nd International Workshop on Advances in Networking and Computing (WANC), pp.271–274, Dec. 2011.

[10] C.J. Colbourn, "Combinatorial aspects of covering arrays," Le Matematiche, vol.59, no.1, 2, pp.125–172, 2004.

[11] M. Grindal, J. Offutt, and S.F. Andler, "Combination testing strategies: A survey," Software Testing, Verification and Reliability, vol.15, no.3, pp.167–199, Sept. 2005.

[12] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Computing Surveys, vol.43, pp.11:1–11:29, Feb. 2011.

[13] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, and C.J. Colbourn, "Constructing test suites for interaction testing," Proc. 25th International Conference on Software Engineering (ICSE'03), pp.38–48, Portland, Oregon, May 2003.

[14] N. Eén and N. Sorensson, "An extensible SAT-solver," Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Lect. Notes Comput. Sci., vol.2919, pp.333–336, 2004.

[15] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "A new method for constructing pair-wise covering designs for software testing," Inf. Process. Lett., vol.81, no.2, pp.85–81, 2002.

[16] J. Czerwonka, "Pairwise testing in real world practical extensions to test case generators," Proc. 24th Annual Pacific Northwest Software Quality Conference, pp.419–430, Oct. 2006.