The Journal of Systems and Software 98 (2014) 191–207

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Generating combinatorial test suite using combinatorial optimization

Zhiqiang Zhang[a,b,d,*], Jun Yan[c,d], Yong Zhao[e], Jian Zhang[a]

[a] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[b] University of Chinese Academy of Sciences, China
[c] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, China
[d] State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, China
[e] Department of Computer Science, College of Engineering and Applied Science, University of Colorado at Colorado Springs, United States

ARTICLE INFO

Article history:
Received 8 November 2013
Received in revised form 13 April 2014
Accepted 1 September 2014
Available online 16 September 2014

Keywords:
Combinatorial testing
Test generation
Combinatorial optimization

ABSTRACT

Combinatorial testing (CT) is an effective technique to test software with multiple configurable parameters. It is used to detect interaction faults caused by the combination effect of parameters. CT test generation aims at generating covering arrays that cover all t-way parameter combinations, where t is a given covering strength. In practical CT usage scenarios, there are usually constraints between parameters, and the performance of existing constraint-handling methods degrades fast when the number of constraints increases.

The contributions of this paper are (1) we propose a new one-test-at-a-time algorithm for CT test generation, which uses pseudo-Boolean optimization to generate each new test case; (2) we have found that pursuing the maximum coverage for each test case is uneconomic, and a possible balance point is to keep the approximation ratio in [0.8,0.9]; (3) we propose a new self-adaptive mechanism to stop the optimization process at a proper time when generating each test case; (4) extensive experimental results show that our algorithm works fine on existing benchmarks, and the constraint-handling ability is better than existing approaches when the number of constraints is large; and (5) we propose a method to translate shielding parameters (a common type of constraints) into normal constraints.

1. Introduction

Combinatorial testing (CT) is used to test software systems with multiple configurable parameters. For real software systems, the number of configurable parameters may be large, and testing all possible configurations is not possible due to limited testing resource. CT enables the tester to execute a small set of test cases on the system, while achieving very high fault coverage.

The first step to apply CT is to build a parameterized model of the system under test (SUT). The tester should first identify the input parameters related to the test goal, i.e. parameters affecting the system behavior which he or she is interested in during the testing process. These parameters may include but not limited to the following:

- Parameters of method calls.
- Parameters in system settings.
- A selection of replaceable system components installed in a test environment, such as hardware devices, system libraries and applications. These components are usually provided by multiple third-party providers, and can have multiple versions.

After the input parameters are identified, the tester needs to identify the value domain of each parameter. CT requires the value domain of each input parameter to be a finite set of discrete values. Typically, the number of possible values should not be too large, or else the test suite size will be very large. If a parameter has too many possible values, the tester should first select several representative values using techniques such as equivalence partitioning, and boundary value analysis.

Sometimes the parameters are not obvious. In these cases, the tester needs to investigate the input domain, and build some *abstract parameters*. Abstract parameters could be some properties of the program input, such as the length of the input file, or the type of a triangle (e.g. is it a right triangle, acute triangle or obtuse triangle, etc.). Then the test cases will be *abstract test cases*, and the tester needs to translate the abstract test cases into *concrete test cases* before execution.

* Corresponding author at: State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China. Tel.: +86 10 62661625.
E-mail addresses: zhangzq@ios.ac.cn (Z. Zhang), yanjun@otcaix.iscas.ac.cn (J. Yan), yzhao@uccs.edu (Y. Zhao), zj@ios.ac.cn (J. Zhang).

http://dx.doi.org/10.1016/j.jss.2014.09.001
0164-1212/© 2014 Elsevier Inc. All rights reserved.

After the modeling process, the SUT is abstracted as a parameterized black-box model consisting of several parameters, and each parameter has a domain of several possible values.

From a black-box perspective, the CT fault model assumes that failures are caused by parameter combinations. These failures are the consequence of *interaction faults* inside the black-box. When several parameters take specific value combinations, a failure occurs. An investigation by Kuhn and Michael (2002) shows that in some systems, failures are usually caused by combinations of size 1–6 in some systems, and about 90% of the failures are caused by parameter combinations of size no more than 3.[1]

The key idea of CT is that since most failures are caused by small parameter combinations, if we have tested all small parameter combinations, then most of the interaction faults can be detected. In CT, we usually use a *covering array* (CA) as the test suite, which covers these small parameter combinations.

There are a lot of researches on covering array generation. The most popular three test generation strategies are the following:

- The one-test-at-a-time strategy first introduced by Cohen et al. (1997) in the Automatic Efficient Test Case Generator (AETG).
- The In-Parameter-Order (IPO) strategy introduced by Lei et al. (2008).
- The strategy of searching for a whole covering array satisfying the coverage criteria, which is used in methods such as EXACT (Yan and Zhang, 2008) and CASA (Garvin et al., 2009, 2011).

In practical CT applications, SUT models usually have constraints between parameters. Every test case must satisfy all parameter constraints, or else it will be invalid and cannot be executed. Ignoring these constraints will make some test cases invalid, and the parameter combinations covered by the invalid test cases may not be tested. Thus some interaction faults supposed to be detected by the test suite may not be detected. Handling constraints is a difficult problem in CT generation. The performance of existing methods degrades fast when the number of constraints increases. Thus better constraint-handling techniques are in great need.

The contributions of this paper are listed as follows: (1) we propose a new one-test-at-a-time algorithm for CT test generation, which uses pseudo-Boolean optimization to generate each new test case; (2) we have found that pursuing the maximum coverage for each test case is uneconomic, and a possible balance point is to keep the approximation ratio in [0.8,0.9]; (3) we propose a new self-adaptive mechanism to stop the optimization process at a proper time when generating each test case; (4) extensive experimental results show that our algorithm works fine on existing benchmarks, and the constraint-handling ability is better than existing approaches when the number of constraints is large; and (5) we propose a method to translate shielding parameters (a common type of constraints) into normal constraints.

This paper is organized as follows: In Section 2, we introduce some background knowledge of CT. In Section 3, we introduce our CT test generation algorithm. In Section 4, we perform experiments to evaluate the algorithm's performance and make discussions. In Section 5 we discuss the related works. Finally, we conclude our work in Section 6.

[1] A recent case study by Ghandehari et al. (2013) revealed that there are some cases where failures are caused by parameter combinations of size greater than 6. The size of the failure-causing parameter combinations depends a lot on the model and the nature of the SUT. However, CT is still applicable in many situations and has been used by a lot of industrial practitioners.

2. Basic concepts and notations

We first introduce some basic concepts used in this paper. An SUT model is defined as follows:

Definition 1. An *SUT model SUT*(P, D) consists of a set of input parameters $P = \{p_1, p_2, \ldots, p_k\}$, and a function D mapping each parameter p_i to its value domain $D(p_i)$. k is called the *number of parameters*, and $s_i = |D(p_i)|$ is called the *level* of parameter p_i.

Definition 2. A *test case* $t = (v_1, v_2, \ldots, v_k)$ is an assignment to all input parameters, such that parameter p_i takes the value of $v_i \in D(p_i)$, for $1 \le i \le k$.

Definition 3. A *combination* $\sigma = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \ldots, (p_{i_l}, v_{i_l})\}$ is an assignment to parameters $p_{i_1}, p_{i_2}, \ldots, p_{i_l}$, such that parameter p_{i_j} takes the value of $v_{i_j} \in D(p_{i_j})$, for $1 \le j \le l$. l is called the *size* of the combination.

A test case $t = (v_1, v_2, \ldots, v_k)$ *covers* combination $\sigma = \{(p_{i_1}, v'_{i_1}), (p_{i_2}, v'_{i_2}), \ldots, (p_{i_l}, v'_{i_l})\}$, if and only if for $1 \le j \le l$, $v_{i_j} = v'_{i_j}$, i.e. the values of $p_{i,j}$ in t and in σ are identical.

A covering array is defined as follows:

Definition 4. A *covering array* $CA(SUT(P, D), t)$ is an $N \times k$ array, where $SUT(P, D)$ is an SUT model, and t is the *strength* of the CA. Each row of the array is a test case, and for $1 \le i \le k$, the ith column corresponds to the values of parameter p_i in test cases. For any t columns of the array, the $N \times t$ sub-array covers all value combinations of the corresponding t parameters.

Note that the original definition of covering arrays requires that all parameters have the same level, however most real SUT models have parameters of different levels. In our paper, a covering array is actually a *mixed-level covering array*, which allows parameters to have different levels. A covering array $CA(SUT(P, D), t)$ can also be denoted as $CA(N; k, (s_1, s_2, \ldots, s_k), t)$, which is a more traditional form, where N is the number of rows (test cases).

The original definition of covering arrays requires all t-way parameter combinations to be covered. However, there are many cases where some parameters interact more (or less) often with each other than with other parameters. If we enforce a global strength, the covering strength needs to be set at the highest interaction level, which will greatly increase the number of test cases, and a lot of resources will be wasted on testing unimportant parameter combinations. Cohen et al. (2003a,b) proposed the concept of *variable strength covering array* (VCA), which allows the tester to specify different covering strengths on different subsets of parameters. Here we use a modified definition:

Definition 5. A variable strength $t^+ = \{(P_1, t_1), (P_2, t_2), \ldots, (P_l, t_l))\}$ is a set of coverage requirements, where P_i is a set of parameters, and t_i is a covering strength on P_i, for $1 \le i \le l$. For $1 \le i \le l$, coverage requirement (P_i, t_i) requires that all t_i-way value combinations of parameters in P_i be covered by the test suite.

When we replace the universal strength t with a variable strength t^+, the covering array will be called a *variable strength covering array*. (Note that the previous definition of the universal strength t can be represented by a variable strength $t^+ = \{(P, t)\}$.) If a variable strength covering array meets covering requirement t^+, then for each $p_i = (P_i, t_i) \in t^+$, the sub-array of parameters in p_i is a covering array of strength t_i.

In the rest of this paper, we use the term *target combinations* to denote parameter combinations which need to be covered as specified by the coverage requirements (covering strength).

Another important concept in CT is *constraints*. Sometimes, some parameters in the SUT model must conform to some restrictions, or else the test case will become invalid. Ignoring parameter

Table 1
An example covering array for a web-based application.

OS	Browser	Flash Player Version	Proxy
Mac OS X	Google Chrome	11	SOCKS5
Mac OS X	Firefox	10	HTTP
Windows	Firefox	9	SOCKS5
Windows	Google Chrome	10	SOCKS4
Mac OS X	Safari	9	SOCKS4
Windows	Internet Explorer	11	HTTP
Linux	Firefox	11	SOCKS4
Linux	Google Chrome	9	HTTP
Linux	Firefox	10	No Proxy
Mac OS X	Safari	11	No Proxy
Windows	Internet Explorer	9	No Proxy
Mac OS X	Safari	10	SOCKS5
Windows	Internet Explorer	10	SOCKS5
Mac OS X	Google Chrome	11	No Proxy
Mac OS X	Safari	11	HTTP
Linux	Firefox	11	SOCKS5
Windows	Internet Explorer	11	SOCKS4

constraints will lead to *coverage holes*, i.e. this will make some test cases to be *invalid* and cannot be executed. If a target combination is only covered by invalid test cases, then the failure caused by this combination will not be detected, since no test cases containing it are executed. Now we give the definition of constraints as follows:

Definition 6. A constraint $c(v_{i_1}, v_{i_2}, \ldots, v_{i_j})$ is a predicate on the assignments of parameters $p_{i_1}, p_{i_2}, \ldots, p_{i_j}$. A test case $t = (v_1, v_2, \ldots, v_k)$ satisfies constraint c if and only if the assignment of those parameters makes the predicate true.

Note that with the presence of constraints, some combinations may violate some of the constraints and cannot occur in any valid test cases, and they are called *invalid*. Also, the coverage requirement will be modified, which requires all valid target combinations be covered by the array.

By adding variable strength and constraints, we change the original covering array definition into $CA(SUT(P, D, C), t^+)$, where $SUT(P, D, C)$ is an extended SUT model containing a set of constraints C, and t^+ is the variable strength coverage requirement.

Let us give a simple example to illustrate how CT works.

Example 1. Suppose we are testing a web application (such as a browser game) to see whether it works on different system configurations. Some possible parameters and their values that can affect the application behaviors are listed as follows:

```
OS: Windows, Linux, Mac OS X;
Browser: Internet Explorer, Firefox, Safari, Google
Chrome;
Flash Player Version: 9, 10, 11;
Proxy: No Proxy, HTTP, SOCKS4, SOCKS5;
```

Since Internet Explorer can only be deployed on Windows, and Safari can only de deployed on Mac OS X, we have the following constraints to forbid invalid configurations:[2]

```
Browser=="Internet Explorer" -> OS=="Windows";
Browser=="Safari" -> OS=="Mac OS X";
```

Now we set the covering strength as 2, then a covering array for this model is shown in Table 1. We can see the array covers all valid combinations between any two parameters, and the above constraints are satisfied.

[2] Internet Explorer for Mac, Internet Explorer for Linux and Safari for Windows have been discontinued. So we assume users rarely use these browsers, and it is reasonable to write these constraints to exclude these rare combinations.

Our previous work (Chen et al., 2010) introduced *shielding parameters*, which describe a special type of constraints that if a certain parameter takes a value in a specific set, then a certain set of other parameters will be *shielded*, i.e. these parameters are ineffective, and their values have no meaning.

Shielding parameters are common in real SUT models especially when some parameters are dependent on some other parameters. For example, suppose we are configuring the parameters of a network interface, if DHCP is enabled, then the manual IP address, the subnet mask and the gateway will be invalid.

Here we consider extending the original concept of shielding parameters to a more general form. We prefer to call this symptom *parameter invalidation*. The formal definition is as follows:

Definition 7. A parameter invalidation constraint is a pair (c, P') where c is a condition, and $P' \subset P$ is a set of parameters to be invalidated when c is true.

The semantics of parameter invalidation constraints is as follows:

- For each parameter p_i, the set of conditions invalidating p_i is:

$$\Phi(p_i) = \{c | (c, P') \text{ is a parameter invalidation constraint and } p_i \in P'\}.$$

- If any condition in $\Phi(p_i)$ is true, then p_i must be invalid.
- If no condition in $\Phi(p_i)$ is true, then p_i must take a valid value.

Thus, we need to know all conditions invalidating each parameter before parameter invalidation constraints can be processed. So we collect all conditions invalidating each parameter, and postpone the processing of these constraints to the very end of the parsing process just before test generation. For each parameter p, an *invalid value* "#" is added. Then we proceed in the following way:

- If no condition invalidates p, then a constraint $\neg(p == \text{"\#"})$ is added.
- If there are some conditions that invalidate p (suppose these conditions are $c_1, c_2, \ldots, c_m$), then the following constraints are added:

$$c_1 \rightarrow (p == \text{"\#"})$$
$$c_2 \rightarrow (p == \text{"\#"})$$
$$\vdots$$
$$c_m \rightarrow (p == \text{"\#"})$$
$$\neg c_1 \wedge \neg c_2 \wedge \cdots \wedge \neg c_m \rightarrow \neg(p == \text{"\#"})$$

3. Covering array generation using combinatorial optimization

In this section, we introduce our new CT test generation technique with constraint-handling support. The goal of covering array generation is to generate an array as small as possible covering all target combinations.

3.1. The covering array generation algorithm

Our covering array generation algorithm is based on the one-test-at-a-time strategy, which was first used in AETG (Cohen et al.,

1997) and has been used in many other algorithms and tools. Algorithm 1 shows the outline of the strategy.

Algorithm 1. The one-test-at-a-time strategy

```
1: $test\_suite = \emptyset$
2: init_and_check($target\_combinations$)
3: while $target\_combinations \neq \emptyset$ do
4:     $new\_test\_case$=gen_new_test_case()
5:     $test\_suite.add(new\_test\_case)$
6:     update($target\_combinations$,$new\_test\_case$)
7: end while
8: return $test\_suite$
```

The algorithm first initializes the set of target combinations, while checking the validity of each combination. All satisfiable target combinations are gathered in *target _combinations*. Then the algorithm enters a while-loop. And in each iteration, it generates a new test case. The goal for generating one new test case, is to maximize the number of newly covered combinations in *target _combinations*. The major difference among one-test-at-a-time algorithms lies here. They use different methods for coverage maximization. Then the algorithm stores the new test case, and updates *target _combinations* by removing the combinations covered in the new test case. The loop terminates until all the satisfiable target combinations are covered.

It is easy to observe that with the presence of constraints, generating each new test case in line 4 is a combinatorial optimization problem, i.e. to find a new test case that satisfies all the constraints while covering as many uncovered combinations as possible.

Our solution to enhance the constraint handling performance is to use an optimization solver to generate new test cases. The previous algorithm can be refined into Algorithm 2.

Algorithm 2. Test generation using combinatorial optimization

```
1:  $test\_suite = \emptyset$
2:  init($target\_combinations$)
3:  while true do
4:      gen_opt_problem()
5:      if solve() == OK then
6:          $new\_test\_case$ = translate_solver_output()
7:          $test\_suite$.add($new\_test\_case$)
8:          update($target\_combinations$, $new\_test\_case$)
9:      else
10:         break
11:     end if
12: end while
```

The main idea of our algorithm is to translate the new test case generation problem into a specific combinatorial optimization problem, use an optimization solver to solve it, and translate the solution back into a test case. In each iteration of the while-loop, it first generates an optimization problem (line 4). Then it calls an optimization solver to solve the problem (line 5). If a solution is successfully found, the algorithm translates the solution into a new test case (line 6), puts it into the test suite (line 7) and updates the set of uncovered combinations (line 8). The loop termination condition is different from Algorithm 1: in line 5, during the solving process, we configure the solver to neglect all solutions covering no combinations in *target _combination*. Thus if no solution is found, it means that no test case can cover any remaining target combination in *target _combinations*. Thus all the remaining target combinations are invalid, and all valid target combinations are already covered. The loop terminates when the newly generated optimization problem is unsatisfiable (line 10).

Our solution will help enhance the performance for constraint handling. The reason is twofold:

1. As a preparation step for generating covering arrays, we need to maintain a set of target combinations. With the presence of constraints, the coverage requirement requires all satisfiable target combinations be covered. So the algorithm needs to know which target combinations are satisfiable and which are not. However, computing all satisfiable target combinations is not an easy task. Suppose the SUT model has 20 parameters of level 3. The number of all possible 3-way parameter combinations is $\binom{20}{3} \times 3^3 = 30,780$. In order to check which combinations are satisfiable, a straightforward way is to check the satisfiability one by one. So the solver needs to be called 30,780 times, which takes a lot of time. Our algorithm forces all test cases cover at least one uncovered target combination, and if no solutions can be found, then all valid target combinations are covered. In this way, the validity check of target combinations in the preparation stage can be avoided and a lot of solver calls will be saved.
2. When assigning parameter values, existing algorithms usually have two levels: the upper algorithm assigns values to parameters in order to cover as many uncovered target combinations as possible, and it usually calls an external constraint solver to check the validity of the assignments or partial assignments. However, the upper algorithm is almost blind about the constraints, and usually does not have good strategies to avoid constraint conflicts since the constraint solver only returns a satisfiable solution or "UNSAT". And when the number of constraint is large, the upper algorithm will more likely encounter constraint conflicts when assigning parameter values, and it will do lots of retries to get a satisfiable solution. Thus a lot of solver calls will be used, and the heuristic for maximizing coverage may not work well since there are too many conflicts to avoid. As for our approach, by encoding the whole new test case generation problem into a combinatorial optimization problem and using an optimization solver, the maximization and the constraint solving are integrated during the assignment process. Thus we are able to avoid conflicts more efficiently when assigning parameters, and the backtrack strategy inside the optimization solver makes it possible to find better solutions.

3.2. *Encoding the problem as a combinatorial optimization problem*

Now we introduce the details of how we translate the problem of generating each new test case into a combinatorial optimization problem. There are many combinatorial optimization problems, such as integer programming and the maximum clique problem. In our work, we translate the problem into a *pseudo-Boolean optimization* (PBO) problem. The reason of choosing PBO is that the problem encoding is relatively straightforward and easy to comprehend. In our implementation, the `clasp` solver (Gebser et al., 2012) is used to solve PBO problems. [3]

A PBO problem consists of a pseudo-Boolean (PB) objective function, and several PB constraints. The variables in the objective function and the constraints are Boolean variables (which can take value of 0 or 1). The PB objective function is a polynomial of PB variables, and the constraints are polynomial relations (equalities and inequalities) of PB variables. And when the objective function and the constraints are linear, the PBO problem is called a *linear PBO problem*. A linear PBO problem is actually identical to the *0-1*

[3] http://potassco.sourceforge.net/.

integer programming problem. An example linear PBO problem in standard form is as follows:

$$\begin{cases} \min : 2x_1 + 3x_3 - x_5 \\ x_1 + 3x_4 + x_5 \geq 2 \\ x_2 + 2x_3 + x_4 = 1 \\ \ldots \end{cases}$$

Before introducing the encoding method, we first give some notations:

- Suppose p_i is a parameter and $v_i \in D(p_i)$ is one possible value of p_i, then we have a PB variable $A_{(p_i,v_i)}$, which is true if and only if p_i takes the value of v_i in the new test case.
- Suppose σ is a combination, then we have a PB variable B_σ corresponding to the coverage of σ, which is true if and only if σ is covered by the new test case.

We encode the new test case generation problem into a PBO problem, which consists of one PB objective function, and three types of PB constraints (Types I–III, which will be shown below). The details are as follows:

3.2.1. The objective function

The objective function is simple, which is encoded as follows:

$$\min : \sum_{\sigma \in target\_combinations} -B_\sigma,$$

thus the number of newly covered target combinations will be maximized. During the solving process, the solver is configured to neglect all solutions whose objective function value is greater than or equal to 0, so that the solver will always find a solution covering at least one uncovered target combination, unless all valid target combinations are covered, and the solver will return an "UNSAT" in this case.

3.2.2. Type I constraints

The first type of PB constraints force all parameters to take exactly one value from their value domain. For each parameter p_i, suppose $D(p_i) = \{d_{i,1}, d_{i,2}, \ldots, d_{i,s_i}\}$, the following PB constraint is encoded:

$$A_{(p_i,d_{i,1})} + A_{(p_i,d_{i,2})} + \cdots + A_{(p_i,d_{i,s_i})} = 1;$$

3.2.3. Type II constraints

This type of PB constraints represents the relation between A-variables and B-variables, i.e. $\sigma = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \ldots, (p_{i_l}, v_{i_l})\}$ is covered, if and only if for all $1 \leq j \leq l$, parameter p_{i_j} takes the value of v_{i_j}. For each $\sigma \in target\_combinations$, a set of PB constraints are encoded as follows:

$$-B_\sigma + A_{(p_{i_1},v_{i_1})} \geq 0;$$

$$-B_\sigma + A_{(p_{i_2},v_{i_2})} \geq 0;$$

$$\ldots$$

$$-B_\sigma + A_{(p_{i_1},v_{i_l})} \geq 0;$$

$$\sum_{1 \leq j \leq l} -A_{(p_{i_j},v_{i_j})} + B_\sigma \geq -l + 1;$$

3.2.4. Type III constraints

This type of PB constraints are translated from the original constraints C in the SUT model. Since the constraints could be in any form, and the solver can only accept PB constraints, we translate the original constraints to PB constraints by using *forbidden combinations* as intermediate form.

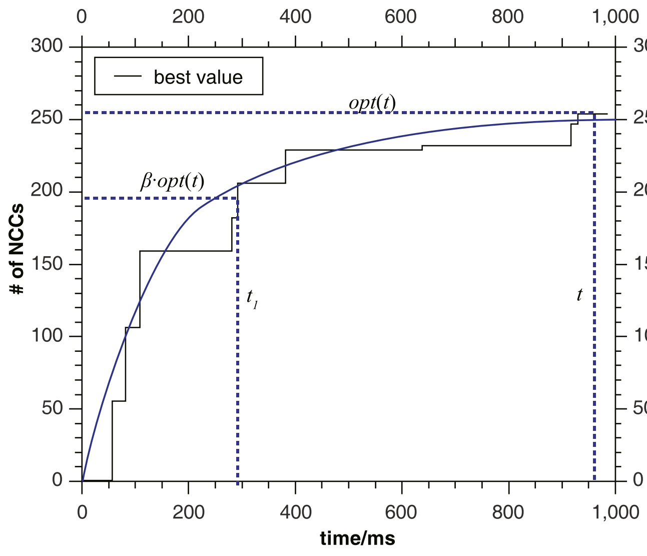

Fig. 1. Growth of the best solution as execution time increases.

For each constraint $c \in C$, let $R(c)$ be the set of all its related parameters. We enumerate all possible combinations of all parameters in $R(c)$. (Since the parameter domains are finite discrete sets, it is possible to enumerate all the combinations, but the cost may grow exponentially with $|R(c)|$.) If any of these combinations is invalid, then it is a *forbidden combination*. It is not hard to understand the original constraint c is equivalent to a constraint that "all forbidden combinations of c must not appear". For each forbidden combination $\sigma = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \ldots, (p_{i_l}, v_{i_l})\}$, we encode it into the following PB constraint:

$$-A_{(p_{i_1},v_{i_1})} - A_{(p_{i_1},v_{i_2})} - \cdots - A_{(p_{i_1},v_{i_l})} >= -l + 1;$$

3.3. Determining solver termination time

If we just pass the generated PBO problem to the solver for optimization, the solver will produce the optimal solution by default, but this will cost it a lot of time. However, practically it is not always necessary to seek for the optimal value. Since our algorithm is based on the one-test-at-a-time strategy, even if each new test case covers the greatest number of uncovered target combinations, there will not necessarily be any significant reduction in the test suite size (which we will explain in Section 4.1).

3.3.1. The solver stopping problem

During the solving process, as time elapses, the current best solution becomes better and better, and improving the solution gets harder and harder. This can be observed in Fig. 1. The line graph in Fig. 1 shows the growth of the current best solution value as execution time increases in a real solving process. The curve profiles the growing speed of the best solution, which gets slower and slower as time increases. Now we want the solver to terminate when the solution is "good enough", so it will not waste too much time on finding the optimal one.

Suppose at a certain point in time t, $opt(t)$ is the number of combinations newly covered by the current best solution (we call the newly covered combinations "*NCCs*" for short in the rest of this paper). We want the solver to terminate when one of the following conditions is met:

- The solving process has already terminated.
- $opt(t)$ is good enough and finding a better solution becomes very hard.

Note that the second condition of solver termination is a premature termination. In this case, the solver is still running, and better solutions may still exist. Determining when to stop the solver is not easy. This is because $opt(t)$ is a step function, and you cannot expect when and how much $opt(t)$ will increase. When $opt(t)$ has not increased for a long period of time (e.g. at 250 ms in Fig. 1), is it a good moment to terminate the solver? There might still be a chance that $opt(t)$ grows significantly after this moment.

Hence we propose a solver stopping mechanism to determine when to stop the solver properly. We first estimate the number of NCCs and the execution time for the next test case to generate according to the history of previously generated test cases. And during the solving process, we assess the quality of the current best solution (denoted as c_1) and the level of difficulty to find better solutions (denoted as c_2). And we combine these two measurements to decide when to stop the solver.

3.3.2. Evaluating the solution quality

The first measurement c_1 evaluates how good the current best solution is. Note that the number of NCCs covered by the optimal solution will gradually decrease as more and more new test cases are generated. From history records, we can obtain how many NCCs are covered by each test case. And in general the number should gradually decrease, too. We use an exponential filter on this number to determine the expected number of NCCs for the next test case:

$$\hat{n}(i+1) = \alpha_n n(i) + (1-\alpha_n)\hat{n}(i), \tag{1}$$

where $n(i)$ and $\hat{n}(i)$ are the observed value and the expected value for the ith test case, and α_n is a *smoothing factor*, which is a constant between 0 and 1. The first measurement c_1 is calculated by the following formula:

$$c_1 = \frac{opt(t)}{\lceil \hat{n}(i) \rceil}. \tag{2}$$

Here "$\lceil x \rceil$" is the ceiling of x, i.e. the smallest integer greater than or equal to x.

3.3.3. Evaluating the solving speed

The second measurement c_2 evaluates how slow the solver is to find better solutions. In our approach, the number of NCCs of the current best solution is recorded along with its corresponding time during the solving process. Suppose at a certain point in time t, the current best solution is $opt(t)$. We locate the earliest time t_1 such that $opt(t_1)$ is greater than or equal to $\beta \cdot opt(t)$, i.e. $t_1 = \min_{t' \geq 0}[opt(t') \geq \beta \cdot opt(t)]$, where β is a constant between 0 and 1. Then we calculate c_2 using the following formula:

$$c_2 = \begin{cases} 0 & t < t_{\min} \\ \dfrac{t}{t_1} & t \geq t_{\min} \end{cases}. \tag{3}$$

Here, $t_{\min}$ is a time threshold, which prevents c_2 to take effect too early (for example we do not want c_2 to take effect before 250 ms in Fig. 1). However there is no universal setting for $t_{\min}$, since the solver may solve larger problems more slowly and require a larger $t_{\min}$. Fortunately, we have a priori knowledge that the PBO problem size for generating the $(i+1)$th test case is about the same with the ith. So we use another exponential filter on the solving time to determine $t_{\min}$ using the following formula:

$$\begin{aligned} \hat{t}(i+1) &= \alpha_t t(i) + (1-\alpha_t)\hat{t}(i) \\ t_{\min}(i+1) &= \max\{\eta \hat{t}(i+1), t^*_{\min}\}, \end{aligned} \tag{4}$$

where $t(i)$ and $\hat{t}(i)$ are the observed time and the expected time for the ith test case, $t_{\min}(i)$ is the minimum time threshold $t_{\min}$ for the ith test case, $t^*_{\min}$ is a universal lower bound of $t_{\min}$, α_t is a smoothing factor between 0 and 1, and η is a factor between 0 and 1.

Table 2
Parameters for solver stopping mechanism.

Parameter	Value
α_n	0.3
α_t	0.3
β	0.8
γ	0.3
θ_1	3.0
θ_2	3.0
$t^*_{\min}$	50 ms
$t_{\min}(1)$	100 ms
η	0.85

3.3.4. Combined evaluation

Now we combine c_1 and c_2 and use the following inequation to determine when to stop the solver:

$$c_1(1+\gamma c_2) \geq \theta_1, \tag{5}$$

where $\gamma > 0$ and $\theta_1 > 1$ are two constant parameters. The inequation has several advantages:

- When the solution is not good enough, c_1 will be small, so the solver will wait longer to try to get a better solution;
- When the solution is relatively good, but the solver falls into local optimum, it may take very long time to find a better solution. In this condition, c_2 will continuously increase and finally let the inequation hold.

3.3.5. Initial settings

Now there is still one unsolved problem: $\hat{n}(1)$ and $t_{\min}(1)$ are missing for generating the first test case. Because our estimation of the problem scale is based on previous solver calls, there is no available information for the first solver call. To overcome this, we use the only c_2 with $t_{\min} = t_{\min}(1)$ for the first solver call:

$$c_2 > \theta_2, \tag{6}$$

where $\theta_2 > 0$ is a constant threshold. Then we set $\hat{n}(2) = n(1)$ and $\hat{t}(2) = 1.5t(1)$. $\hat{t}(2)$ is set as $1.5t(1)$ because the first solver call might approach global optimum very quickly since any feasible solution will cover a lot of uncovered combinations, so the solver may terminate very early using (6). Noticing this fact, we multiply $t(1)$ with a factor of 1.5. After this, the stopping mechanism can work as we mentioned before in (1)–(5).

In our implementation, all the parameters mentioned above are set as shown in Table 2. These parameter values are finely tuned, which we will show in Section 4.5.

4. Evaluation

In this section, we conduct some experiments to evaluate the performance of our approach. We propose four research questions, and then we use experimental data to answer them. The research questions are as follows:

- **RQ1:** Are optimal solutions for each test case necessary? If not, how less the number of NCCs of each test case could be, without significantly affecting the test suite size?
- **RQ2:** Does our stopping mechanism work well?
- **RQ3:** What is the performance of our approach w.r.t. the execution time and the test suite size?
- **RQ4:** Can our approach handle parameter constraints well?

And afterwards, we will discuss the parameters of our solver stopping mechanism with experimental results.

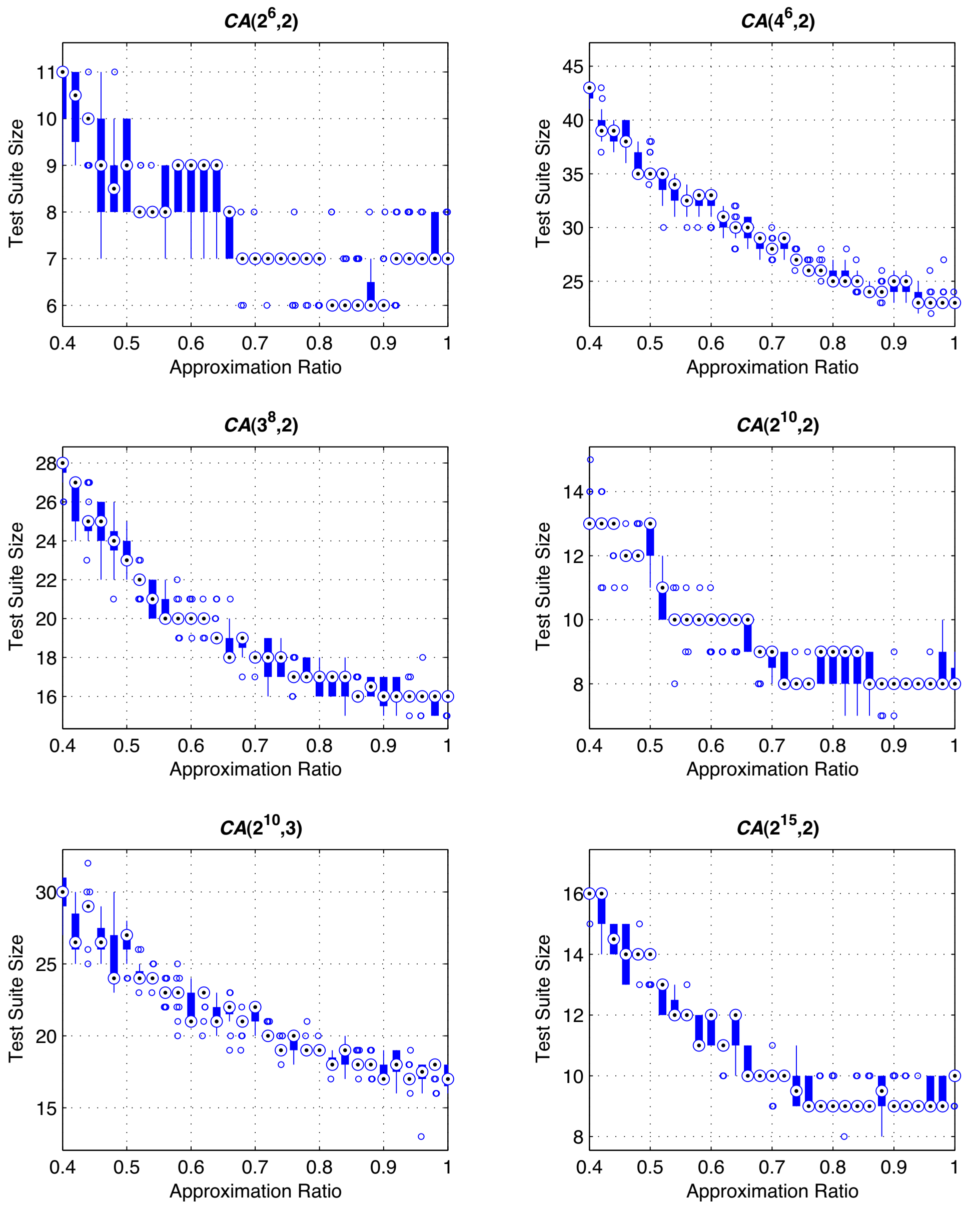

Fig. 2. Influence of the approximation ratio on the test suite size.

In this section, we use a short form representation for covering arrays. For example, $CA(4^6, 2)$ denotes a 2-way covering array of 6 parameters of level 4.

4.1. RQ1: Are optimal solutions for each test case necessary?

To answer this question, we configure our test generator to generate each test case in the following way:

- First, it generates a test case without using the stopping mechanism, so each solution is optimal. The number of NCCs is counted as n^*.
- Then we multiply n^* with a factor $0 < r \le 1$, and call the solver once again using the stopping mechanism, with $\hat{n} = \lceil rn^* \rceil$, $\gamma = 0$ and $\theta = 1$. So the solver will terminate exactly when the solution reaches $\lceil rn^* \rceil$. Here r is called the *approximation ratio*, and the solution is called *"r-approximate"*.

In this way we generate test suites for several models, and we vary the approximation ratio to see how the test suite size changes. Fig. 2 shows the test suite size distribution for six SUT models, at different approximation ratios in [0.4, 1.0]. The results are taken from 20 random runs. We normalize the mean test suite size by dividing it by the test suite size at $r = 1$. The influence of the approximation ratio on the normalized mean test suite size is shown in Fig. 3.

In general, when the approximation ratio increases, the test suite size will get smaller. When r reaches around 0.7, the test suite size will be smaller than 1.2 times of the test suite size at $r = 1$. When r reaches around 0.8, the test suite will be smaller than 1.1 times of the test suite size at $r = 1$.

Since finding the optimal solutions is NP-hard, we did not perform this experiment on larger SUT models. However the results still provide evidence that if the approximation ratio reaches about 0.8, the percentage difference of the test suite size with the test suite size at $r = 1$ is within 10%. Additionally, for some of the models, the test suite size at $r = 1$ is not the lowest point. In these cases, the test

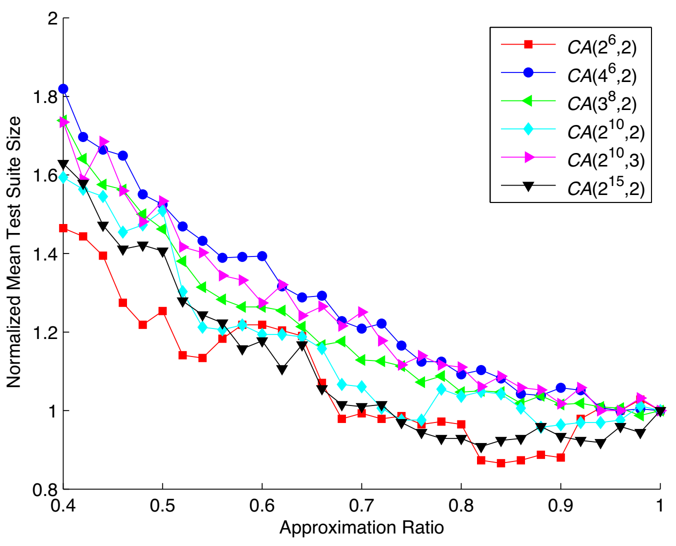

Fig. 3. Influence of the approximation ratio on the test suite size (normalized).

cases are *over-optimized*. Now this shows that it is uneconomic to achieve maximum coverage for each test case in the one-test-at-a-time strategy. Observed from Fig. 3, a good approximation ratio should be around 0.8–0.9. (For large SUT models, we still need more evidence for a proper approximation ratio.)

4.2. RQ2: Does our stopping mechanism work well?

Our stopping mechanism aims at balancing the quality of the solution and the execution time.

Fig. 4 shows the number of NCCs and the solving time for each test case for using and not using the stopping mechanism for some small instances. We can see that by using our stopping mechanism, the number of NCCs is very close to the case of not using it, while the solving time for each test case is greatly reduced.

Fig. 5 shows the data on some large instances. (Since it takes too much time for running our algorithm on large instances without the stopping mechanism, we only show the results for using the stopping mechanism.) The number of NCCs decreases smoothly except for some small jitters. This shows our stopping mechanism is still stable and robust for large SUT models.

Besides, we can observe in Fig. 5 that it is relatively easy to reach a near-optimal solution in the beginning and the tailing solver calls. The most difficult part is in the middle area. The reason is as follows: (1) in the beginning, most target combinations are uncovered, and it is very easy to avoid the covered target combinations and reach a high coverage; (2) during the middle area, the uncovered target combinations become scattered. As a consequence, many conflicts will occur when choosing target combinations to cover. (3) In the end, most target combinations are covered, and the remaining target combinations are very likely to have conflicts with each other, so the test case can only cover a few target combinations at most.

However, there is a threat to validity to our stopping mechanism. Look at the sub-figure for $CA(2^{50}, 2)$ in Fig. 5. The solving time increases extremely fast for the first several test cases. This phenomenon is more severe when we experiment with larger models such as $CA(2^{100}, 2)$. The reason is that for this kind of large models with small parameter levels, there are not too many test cases needed to meet the coverage requirement, and the number of NCCs for each test case drops extremely fast. Since we did not predict how fast the number drops, we only assume that the number of NCCs is close to the last several solver calls. The number does not reach the expected value, so our stopping mechanism determines that the problem is very hard, and waits for a very long period of time to reach the expected value. A possible work around to get rid of this phenomenon is to refine the exponential filter that calculates the expected number of NCCs by adding a prediction mechanism that evaluates how fast the number of NCCs decreases.

Table 3
SUT model details.

Model	Description	# constraints
Apache	$CA(2^{158}3^{8}4^{4}5^{1}6^{1}\,;2)$	7
Bugzilla	$CA(2^{49}3^{1}4^{2}\,;2)$	5
gcc	$CA(2^{189}3^{10}\,;2)$	40
SpinS	$CA(2^{13}4^{5}\,;2)$	13
SpinV	$CA(2^{42}3^{2}4^{11}\,;2)$	49
Banking1	$CA(3^{4}4^{1}\,;2)$	112
Banking2	$CA(2^{14}4^{1}\,;2)$	3
CommProtocol	$CA(2^{10}7^{1}\,;2)$	128
Concurrency	$CA(2^{5}\,;2)$	7
Healthcare1	$CA(2^{6}3^{2}5^{1}6^{1}\,;2)$	21
Healthcare2	$CA(2^{5}3^{6}4^{1}\,;2)$	25
Healthcare3	$CA(2^{16}3^{6}4^{5}5^{1}6^{1}\,;2)$	31
Healthcare4	$CA(2^{13}3^{12}4^{6}5^{2}6^{1}7^{1}\,;2)$	22
Insurance	$CA(2^{6}3^{1}5^{1}6^{2}11^{1}13^{1}17^{1}31^{1}\,;2)$	0
NetworkMgmt	$CA(2^{2}4^{1}5^{3}10^{2}11^{1}\,;2)$	20
ProcessorComm1	$CA(2^{3}3^{6}4^{6}\,;2)$	13
ProcessorComm2	$CA(2^{3}3^{12}4^{8}5^{2}\,;2)$	125
Services	$CA(2^{3}3^{4}5^{2}8^{2}10^{2}\,;2)$	388
Storage1	$CA(2^{1}3^{1}4^{1}5^{1}\,;2)$	95
Storage2	$CA(3^{4}6^{1}\,;2)$	0
Storage3	$CA(2^{9}3^{1}5^{3}6^{1}8^{1}\,;2)$	48
Storage4	$CA(2^{5}3^{7}4^{1}5^{2}6^{2}7^{1}10^{1}13^{1}\,;2)$	24
Storage5	$CA(2^{5}3^{8}5^{3}6^{2}8^{1}9^{1}10^{2}11^{1}\,;2)$	151
SystemMgmt	$CA(2^{5}3^{4}5^{1}\,;2)$	17
Telecom	$CA(2^{5}3^{1}4^{2}5^{1}6^{1}\,;2)$	21
CA_1	$CA(3^{4}\,;2)$	0
CA_2	$CA(3^{10}\,;2)$	0
CA_3	$CA(3^{20}\,;2)$	0
CA_4	$CA(2^{4}\,;2)$	0
CA_5	$CA(2^{10}\,;2)$	0
CA_6	$CA(2^{50}\,;2)$	0
CA_7	$CA(5^{6}\,;3)$	0
CA_8	$CA(5^{10}\,;3)$	0
CA_9	$CA(2^{10}\,;4)$	0
CA_10	$CA(2^{10}\,;5)$	0
CA_11	$CA(2^{10}\,;6)$	0
VCA_1	$VCA(3^{15}\,;2,\{CA(3^{5}\,;3)\})$	0
VCA_2	$VCA(3^{15}\,;2,\{CA(3^{5}\,;3), CA(3^{5}\,;3), CA(3^{5}\,;3)\})$	0
VCA_3	$VCA(4^{3}5^{6}6^{6}\,;2,\{CA(4^{3}\,;3)\})$	0
VCA_4	$VCA(4^{3}5^{6}6^{6}\,;2,\{CA(4^{3}5^{3}6^{1}\,;3)\})$	0

4.3. RQ3: What is the performance of our approach w.r.t. the test generation time and the test suite size?

To answer this question, we compare our algorithm (implemented in our Cascade tool (Zhao et al., 2013)) with a selection of existing tools (ACTS (Lei et al., 2008), PICT (Czerwonka, 2006) and CASA (Garvin et al., 2009, 2011)) on several benchmark SUT models. The details of these benchmark models are as shown in Table 3. The first 5 models are taken from the 35 models by Cohen et al. (2008). (These 5 models are constructed from real case studies. And the other 30 models are synthesized based on the characteristics of the 5 real models. Here we only use the 5 real models.) The models from row “Banking1” to row “Telecom” are taken from a paper by Segall et al. (2011) as whole. The models from row “CA_1” to “CA_11” are some general models varying from very simple models to hard ones. The models from row “VCA_1” to “VCA_4” are taken partly from a paper by Cohen et al. (2003b). In our approach, the treating for VCAs is actually the same with the treating for normal CAs (since the coverage requirements are both translated into target combinations), so we only include 4 of the VCA benchmarks. In Table 3, the model descriptions are in short form representations which are used in many other works on CT. For example, $CA(2^{49}3^{1}4^{2}\,;2)$ means a covering array at strength 2 of an SUT having 49

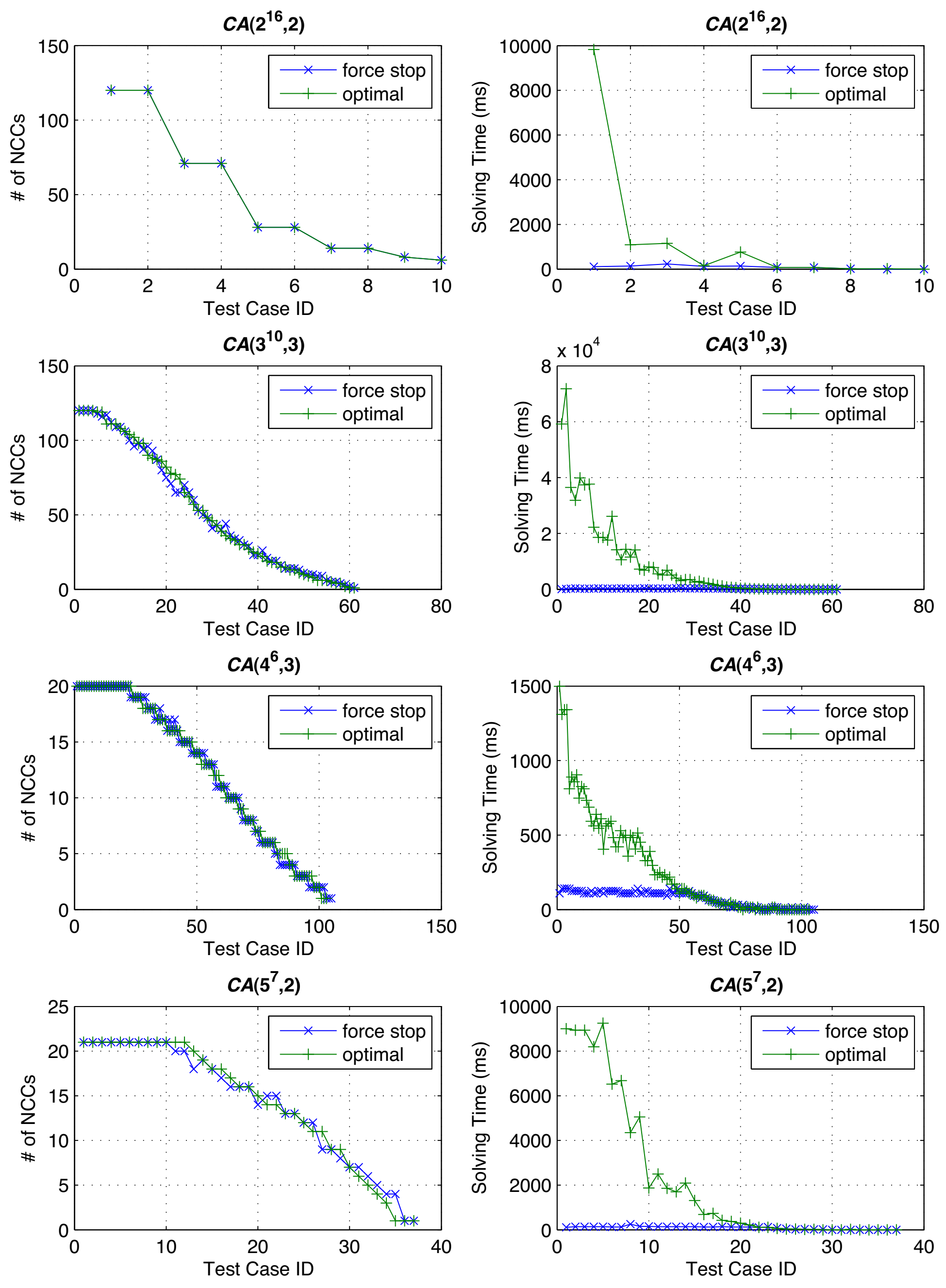

Fig. 4. Using vs. not using our stopping mechanism.

parameters of level 2, 1 parameter of level 3 and 2 parameters of level 4. $VCA(3^{15}; 2, \{CA(3^5; 3), CA(3^5; 3), CA(3^5; 3)\})$ is a variable strength covering array, which has a 2-way coverage requirement on 15 parameters of level 3, and three 3-way coverage requirements on 3 disjoint sets of 5 parameters of level 3. The results are shown in Tables 4 and 5. The results are taken from 20 random executions for each SUT model and tool (except for ACTS which is deterministic). All the experiments are performed on a 64-bit Windows 7 machine with Intel Core i5 CPU (M540 2.53 GHz) and 4 GB memory.

From the results in Tables 4 and 5 we have the following observations and explanations:

- The size of covering arrays generated by our tool is comparable with the other three tools.
- CASA uses a different search strategy, which applies simulated annealing on the whole covering array instead of extending the covering array vertically or horizontally. The strategy enables CASA to find some smaller covering arrays than the other three tools.
- CASA does not support VCAs, so the corresponding entries are labeled “NA”.
- The test suite size for PICT on VCAs is much larger than other algorithms. The reason is that it uses hierarchical sub-models to define VCAs, so we tag these entries with asterisks (*). PICT first generates covering arrays for sub-models. Then it treats the sub-model as a new parameter and all the test cases for this sub-model as the values of the new parameter, in order to construct covering arrays for the parent model.
- The execution time of our tool is relatively long, and in some cases the algorithm even failed to complete within the time out limit. Since our implementation uses a universal solver for PBO problems, it may be slower than algorithms using heuristics

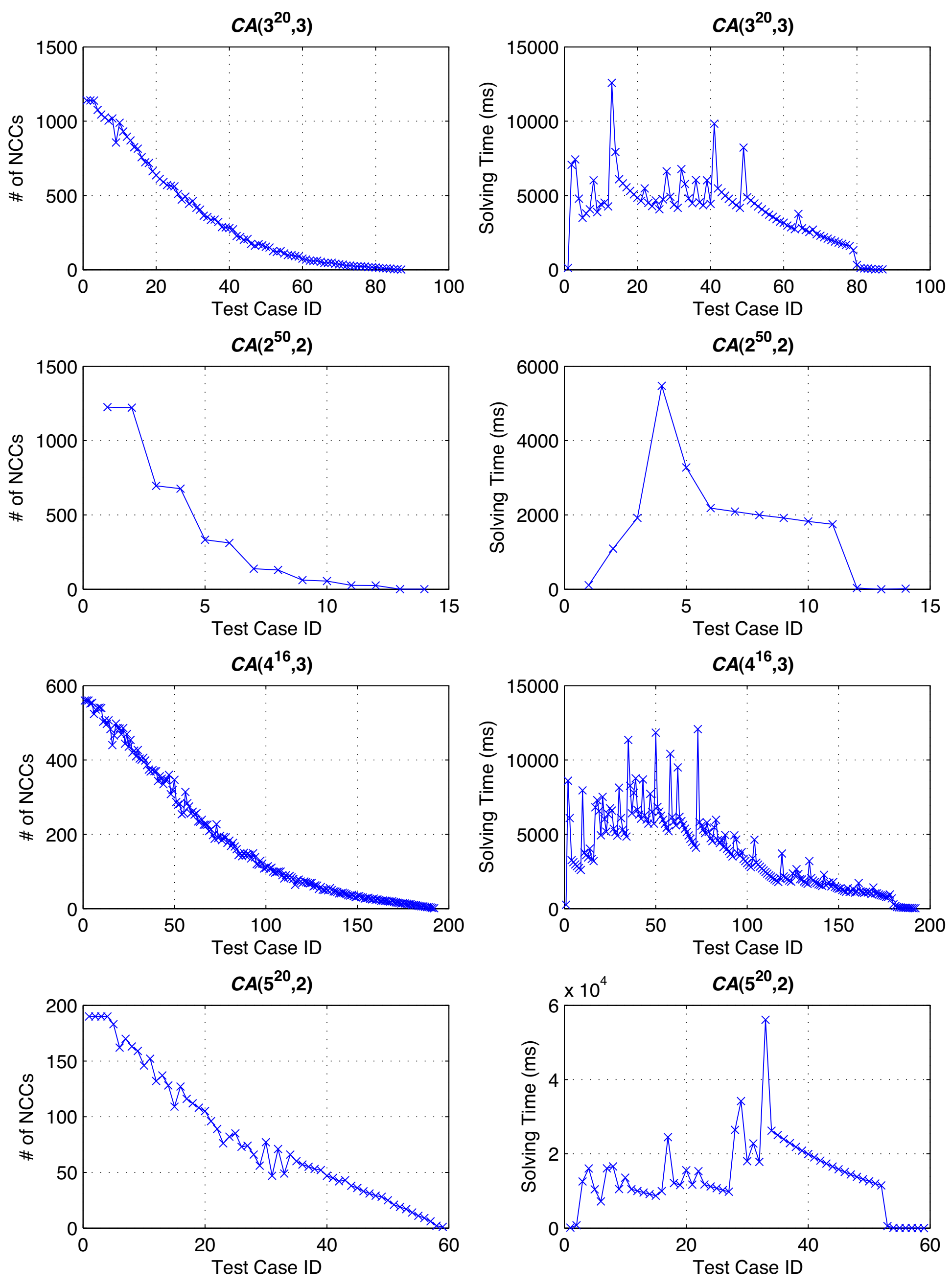

Fig. 5. Stopping mechanism performance for large models.

for parameter assignment. Also, the execution time of our tool increases fast when the SUT model gets larger. The reason is that the size of the generated PBO problem increases exponentially with the SUT model size.

- For the models with large number of constraints (model names in **bold** font). We can see that the execution time of our tool is comparable with or even shorter than other algorithms. This gives us a small peek at the advantage of our algorithm on constraint handling, which we will discuss in answering RQ4.

4.4. RQ4: Can our approach handle parameter constraints well?

To answer this question, we fix a base covering array, say $CA(2^8, 2)$. We generate various number of constraints and evaluate the algorithm performance with respect to the number of constraints. In our evaluation, all the constraints are in the forbidden combinations form, and two sets of constraints are generated:

- The first set of forbidden combinations are randomly generated. Here we randomly generate a given number of forbidden combinations. Each forbidden combination is constructed by randomly selecting 3 parameters and one value for each parameter.
- The second set of forbidden combinations are relatively more structured and easier to solve compared with random constraints. We randomly select 5 parameters, pick one value for each parameter, and generate several constraints that forbid more than 2 parameters to take the selected values. For example, suppose the selected parameters are $(p_{i_1}, p_{i_2}, p_{i_3}, p_{i_4}, p_{i_5})$ and the selected values are $(v_{j_1}, v_{j_2}, v_{j_3}, v_{j_4}, v_{j_5})$. Then the following 10 constraints are generated:

$$\begin{array}{l}
p_{i_1} \neq v_{j_1} \vee p_{i_2} \neq v_{j_2} \vee p_{i_3} \neq v_{j_3} \\
p_{i_1} \neq v_{j_1} \vee p_{i_2} \neq v_{j_2} \vee p_{i_4} \neq v_{j_4} \\
p_{i_1} \neq v_{j_1} \vee p_{i_2} \neq v_{j_2} \vee p_{i_5} \neq v_{j_5} \\
p_{i_1} \neq v_{j_1} \vee p_{i_3} \neq v_{j_3} \vee p_{i_4} \neq v_{j_4} \\
p_{i_1} \neq v_{j_1} \vee p_{i_3} \neq v_{j_3} \vee p_{i_5} \neq v_{j_5} \\
p_{i_1} \neq v_{j_1} \vee p_{i_4} \neq v_{j_4} \vee p_{i_5} \neq v_{j_5} \\
p_{i_2} \neq v_{j_2} \vee p_{i_3} \neq v_{j_3} \vee p_{i_4} \neq v_{j_4} \\
p_{i_2} \neq v_{j_2} \vee p_{i_3} \neq v_{j_3} \vee p_{i_5} \neq v_{j_5} \\
p_{i_2} \neq v_{j_2} \vee p_{i_4} \neq v_{j_4} \vee p_{i_5} \neq v_{j_5} \\
p_{i_3} \neq v_{j_3} \vee p_{i_4} \neq v_{j_4} \vee p_{i_5} \neq v_{j_5}
\end{array}$$

The results are shown in Fig. 6. First, we compare the average execution time of each tool for generating each test case, which is proportional to the average time needed for each effective assignment, and reflects the constraint-handling efficiency. From sub-figures (c) and (f), we can see that the average execution time for each test case of our approach is relatively long when the number of constraints is small, but it is very insensitive to the increase of the number of constraints. By contrast, the execution time of all the other three tools increases as the number of constraints increases. Generally, the order of sensitivity to the number of constraints is as follows: PICT > ACTS > CASA > Cascade (our approach).

Then we compare the test suite size (shown in sub-figures (b) and (e)). In most cases, CASA can still generate the smallest test suite, and Cascade works better than ACTS and PICT. However, as shown in sub-figure (b), when the number of random constraints reaches 640, the test suite size drops to around 6 for our approach, ACTS and PICT, but the test suite size for CASA is still around 27. At this point, the constraints are so extensive that only a few test cases are valid, and only a few target combinations are valid. So the test suite size decreases to 6. As for CASA, its implementation of simulated annealing tries to improve the coverage of the current array by randomly replacing an entry in the array, and the new array is abandoned if it violates the constraints. So when the constraints are extensive enough, this strategy can hardly jump from one satisfiable solution to a better satisfiable solution. So CASA is unlikely to generate small covering arrays in this situation.

4.5. Parameter tuning

Now we discuss the parameters for our solver stopping mechanism. The ideal goal is to find a parameter setting suitable for all CT test generation problems. However this is not possible. Besides, we also should make a balance between reducing the test suite size and reducing the test generation time, which are two conflicting goals. Here we decide to choose several medium-sized CT test generation problems, find a good parameter setting for them, and use it as the parameter setting throughout this paper. The problems used for parameter tuning are $CA(4^8, 3)$, $CA(3^{10}, 3)$, $CA(2^{40}, 2)$ and $CA(5^{16}, 2)$.

There are 9 parameters that need to be set, thus the number of possible settings is too large. So we reduce the number of parameters that need to be tuned. In our work, α_n and α_t are equivalent. Besides, parameter θ_2, $t^*_{\min}$ and $t_{\min}(1)$ actually take very little effect. So there are 5 parameters that need to be tuned in total. However it is still difficult to find a good parameter setting. After many attempts, we found a balanced parameter setting as shown in Table 2, although it may not be the best one.

Here, we vary the value of each parameter while keeping the values of other parameters fixed to illustrate its influence on the test suite size and the execution time. The results are shown in Figs. 7–11. The results are taken from the mean value of 20 random runs, and are then normalized by dividing by the standard value, which is the mean value when the parameters take values shown in Table 2.

Here we make explanations about the results:

- By intuition, more execution time may lead to smaller test suite size. However, the results for $CA(2^{40}, 2)$ is against this intuition. So we deduce that test cases for this model are over-optimized.
- The smoothing factors α_n and α_t specifies how fast the expected time and the number of NCCs changes with the observed ones. With a smaller smoothing factor, the expected time and the number of NCCs decreases slower, so that it may find better solutions for each test case, but the execution time may be longer. However, since the time and the number of NCCs for each test case is almost the same (except for $CA(2^{40}, 2)$), the smoothing factor actually has less influence than the other four parameters. From Fig. 7, we can see the small variations of the test suite size have no clear tendency and are within 2%, which is likely to be noises. Being relatively conservative, we set the smoothing factor as 0.3.
- Parameter β is used to determine how slowly the solver finds a better solution. For a certain optimization process, at a certain point in time, the greater β is, the smaller c_2 will be. Thus according to (5), as β increases, the number of NCCs and the execution time for each test case will increase. So the test suite size will decrease (except for $CA(2^{40}, 2)$ which is over-optimized). And when the solution of each test case is near-optimal, finding a better solution needs more time, and the improvement may be very small. Thus the decrease of the test suite size cannot supplement the increase of the execution time for each test case, so the total execution time will increase. From Fig. 8, we can see 0.8 is a good balance point for β for minimizing the test suite size and minimizing the execution time.
- Parameter γ specifies how much c_2 takes part in the stopping condition. When γ increases, the left hand side of (5) increases, allowing the solver to stop earlier. Thus the number of NCCs and the execution time for each test case will decrease. Similar reason as β, the test suite size increases, and the total execution time decreases (except for $CA(2^{40}, 2)$ which is over-optimized). From Fig. 9, we can see 0.3 is a good balance point for β for minimizing the test suite size and minimizing the execution time.
- Parameter θ_1 specifies the right hand side of (5). When θ_1 increases, the number of NCCs and the execution time for each test case will decrease. Similar with β and γ, the test suite size increases, and the total execution time decreases. From Fig. 10, we can see 3.0 is a good balance point for θ_1 for minimizing the test suite size and minimizing the execution time.
- Parameter η specifies how long before the solver stopping mechanism works. From (4) we know that the mechanism starts to work at least from $\eta\hat{t}(i+1)$, where $\hat{t}(i+1)$ is the expected execution time of the current test case. If η is too small, the stopping mechanism may start to work too early. And since c_2 may be quite large in the beginning, the solver may stop too early so better solutions may be missed. Thus the test suite size may be large. However if η is too large, when the PBO problem for each test case becomes easier and needs less time to solve, the solver may stop too late and time will be wasted. When η is not too large or too small, it has little influence on the test suite size and the execution time. These can be observed from Fig. 11. So we select value 0.85 for η, which makes the execution time just start to increase significantly.

5. Related works

Combinatorial testing (CT) has been researched for many years. Nie and Leung (2011) provided a rich survey of existing works on some major research topics of CT. Kuhn et al. (2013) wrote a book, which covers many theoretical and practical aspects of CT.

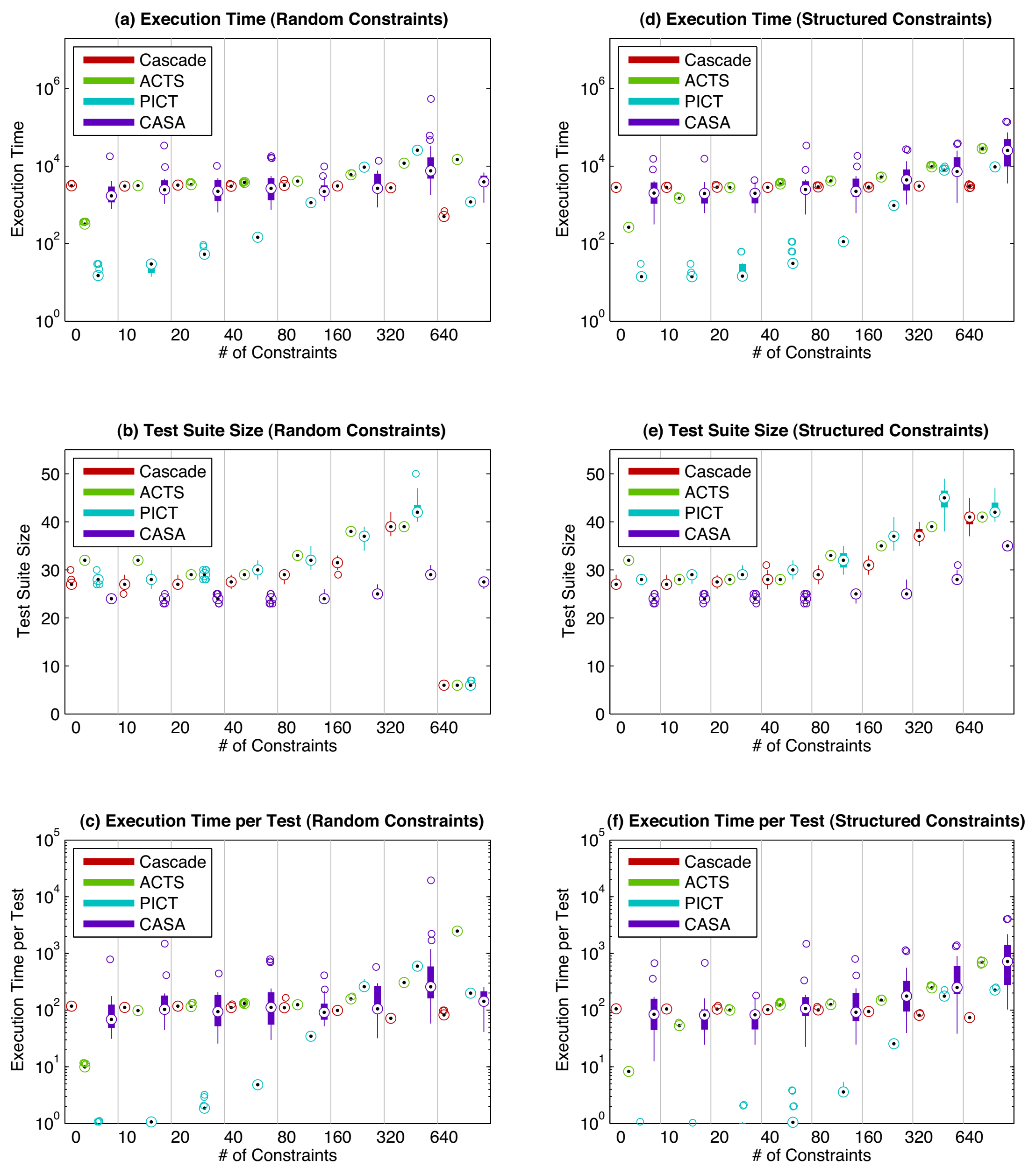

Fig. 6. Influence of constraints on performance.

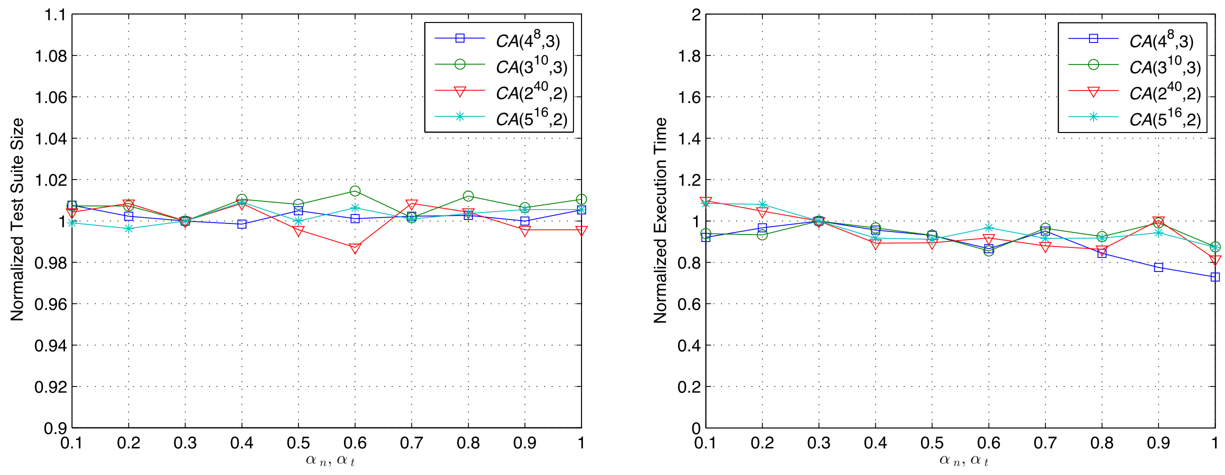

Fig. 7. Influence of α_n and α_t on the test suite size and execution time.

Table 4
Comparison of test suite size.

Subjects	Cascade				ACTS				PICT				CASA			
	Avg.	Min.	Max.	*#TO*	Avg.	Min.	Max.	#TO	Avg.	Min.	Max.	*#TO*	Avg.	Min.	Max.	#TO
Apache	NA	NA	NA	20	33.0	33	33	0	39.9	37	43	0	33.7	30	37	0
Bugzilla	21.8	20	25	0	18.0	18	18	0	19.8	17	22	0	17.1	16	21	0
gcc	NA	NA	NA	20	25.0	25	25	0	32.3	27	35	0	22.0	18	28	3
SpinS	27.0	26	28	0	27.0	27	27	0	26.4	25	28	0	19.8	19	22	0
SpinV	43.2	39	48	1	46.0	46	46	0	63.2	57	70	0	41.0	34	47	0
Banking1	15.2	14	17	0	15.0	15	15	0	16.4	14	19	0	12.1	12	13	0
Banking2	12.9	11	14	0	11.0	11	11	0	13.0	12	14	0	10.0	10	10	0
CommProtocol	19.1	17	21	0	19.0	19	19	0	20.3	18	23	0	20.6	19	22	0
Concurrency	5.7	5	6	0	6.0	6	6	0	5.7	5	7	0	6.0	6	6	0
Healthcare1	32.1	30	34	0	30.0	30	30	0	30.9	30	33	0	30.0	30	30	0
Healthcare2	18.3	17	20	0	16.0	16	16	0	18.9	17	21	0	14.8	14	15	0
Healthcare3	39.9	38	44	0	38.0	38	38	0	41.2	38	45	0	31.2	30	35	0
Healthcare4	55.5	52	58	0	49.0	49	49	0	53.7	50	59	0	43.0	42	48	0
Insurance	531.7	528	537	0	527.0	527	527	0	527.0	527	527	0	539.0	527	550	0
NetworkMgmt	120.9	118	125	0	112.0	112	112	0	125.7	122	132	0	115.2	110	122	0
ProcessorComm1	29.7	27	32	0	29.0	29	29	0	28.9	27	30	0	27.6	26	30	0
ProcessorComm2	34.5	31	38	0	32.0	32	32	0	34.9	33	38	0	29.6	27	34	0
Services	109.7	106	113	0	106.0	106	106	0	112.4	109	116	0	105.3	102	110	0
Storage1	18.0	17	19	0	17.0	17	17	0	18.3	17	20	0	20.0	20	20	0
Storage2	19.8	18	21	0	18.0	18	18	0	20.3	18	22	0	18.0	18	18	0
Storage3	53.5	51	57	0	50.0	50	50	0	54.6	52	58	0	44.5	44	46	0
Storage4	136.9	133	144	0	136.0	136	136	0	130.1	130	131	0	131.1	130	133	0
Storage5	238.7	232	246	0	218.0	218	218	0	235.1	231	240	0	159.4	155	168	0
SystemMgmt	18.6	17	20	0	17.0	17	17	0	19.1	18	20	0	17.0	17	17	0
Telecom	32.4	31	34	0	32.0	32	32	0	31.2	30	33	0	31.3	31	33	0
CA_1	10.1	9	12	0	9.0	9	9	0	11.1	9	13	0	9.0	9	9	0
CA_2	17.3	16	19	0	15.0	15	15	0	17.9	17	19	0	15.0	15	15	0
CA_3	21.6	20	23	0	24.0	24	24	0	22.6	21	24	0	18.1	17	19	0
CA_4	6.0	6	6	0	6.0	6	6	0	5.1	5	6	0	5.0	5	5	0
CA_5	8.7	8	9	0	10.0	10	10	0	8.3	8	9	0	6.0	6	6	0
CA_6	12.1	12	13	0	14.0	14	14	0	14.0	14	14	0	9.2	9	10	0
CA_7	207.9	204	214	0	201.0	201	201	0	216.2	211	221	0	186.1	177	209	5
CA_8	NA	NA	NA	20	307.0	307	307	0	306.4	303	309	0	271.0	267	277	9
CA_9	38.8	35	42	0	44.0	44	44	0	42.0	39	44	0	24.7	24	38	0
CA_10	83.5	81	86	0	93.0	93	93	0	88.2	85	91	0	71.8	56	77	2
CA_11	NA	NA	NA	20	178.0	178	178	0	171.0	167	175	0	144.5	141	153	4
VCA_1	40.5	38	43	0	41.0	41	41	0	125.1*	117*	134*	0	NA	NA	NA	NA
VCA_2	47.3	45	49	0	48.0	48	48	0	1972.8*	1833*	2100*	0	NA	NA	NA	NA
VCA_3	69.1	66	75	0	70.0	70	70	0	386.7*	384*	392*	0	NA	NA	NA	NA
VCA_4	217.2	212	221	1	NA	NA	NA	20	1266.1*	1231*	1293*	0	NA	NA	NA	NA

Here we first discuss constraint-handling techniques of CT test generation algorithms. The problem was recognized very early by researchers. Tatsumi et al. (1987) mentioned that special consideration should be taken for constraints when generating covering arrays. The most natural form for representing constraints is to use predicates on parameter values, which is easy for human writing and comprehension. Hartman and Raskin (2004) formalized constraints into forbidden configurations, which are test cases that should not appear in the test suite. And then many works used forbidden combinations for representing and handling constraints more conveniently (Bryce and Colbourn, 2006; Czerwonka, 2006; Cohen et al., 2008). These forbidden combinations are usually

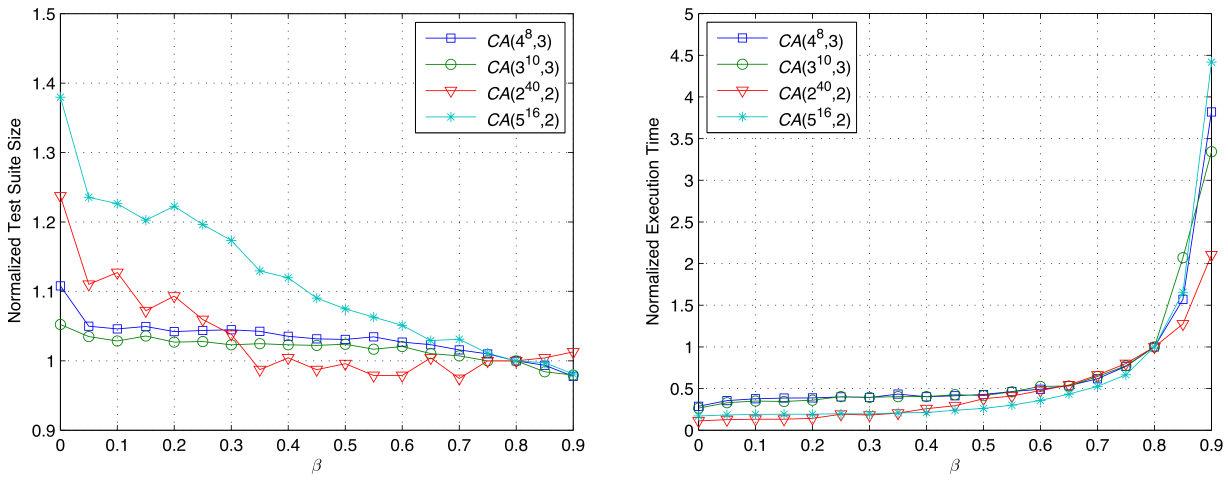

Fig. 8. Influence of β on the test suite size and execution time.

Table 5
Comparison of execution time (in milliseconds).

	Cascade				ACTS (IPOG-C)				PICT				CASA (simulated annealing)			
Subjects	Avg.	Min.	Max.	#TO	Avg.	Min.	Max.	#TO	Avg.	Min.	Max.	#TO	Avg.	Min.	Max.	#TO
Apache	NA	NA	NA	20	2628.3	2414	3004	0	218.3	212	263	0	75,898.9	35,256	204,363	0
Bugzilla	153,954.0	74,609	284,196	0	1278.6	1177	1580	0	30.5	26	40	0	5479.9	2368	19,165	0
gcc	NA	NA	NA	20	4654.5	4256	5131	0	250.1	212	266	0	700,239.1	55,242	1,612,454	3
SpinS	4984.9	3670	7037	0	2077.4	2016	2140	0	25.7	14	32	0	10,953.3	663	50,403	0
SpinV	1,037,832.7	589,868	1,779,312	1	4459.9	4063	4653	0	65.3	62	113	0	89,578.3	9476	361,404	0
Banking1	686.1	593	809	0	3906.7	3712	4521	0	33.6	30	63	0	537.0	196	1044	0
Banking2	1251.8	1146	1447	0	689.3	631	759	0	15.8	14	31	0	403.0	341	791	0
CommProtocol	899.3	845	991	0	6490.4	6205	6803	0	712.8	613	863	0	1047.4	491	2313	0
Concurrency	188.6	143	249	0	854.7	811	912	0	14.4	14	16	0	153.2	140	197	0
Healthcare1	1830.8	1715	2107	0	1637.9	1574	1709	0	17.8	14	34	0	428.5	366	702	0
Healthcare2	1872.3	1716	2101	0	2935.3	2732	3389	0	32.0	29	62	0	1059.8	345	2898	0
Healthcare3	138,532.8	59,715	262,228	0	4733.0	4591	4966	0	30.8	30	35	0	6050.2	3033	12,860	0
Healthcare4	417,341.9	267,744	649,523	0	4590.1	4459	4824	0	56.6	30	66	0	19,478.8	4627	88,677	0
Insurance	286,208.1	227,168	350,130	0	506.6	448	574	0	112.7	112	118	0	30,666.8	11,482	258,089	0
NetworkMgmt	15,929.6	12,814	19,125	0	2373.1	2307	2486	0	30.6	30	34	0	41,328.8	1560	165,253	0
ProcessorComm1	6119.9	4410	14,552	0	1604.1	1539	1685	0	23.3	14	33	0	2463.9	571	11,239	0
ProcessorComm2	198,232.7	49,301	477,768	0	6961.0	6733	7242	0	33.5	30	64	0	16,278.5	1188	57,158	0
Services	28,169.7	22,428	43,597	0	18,475.0	17,339	19,681	0	316.9	312	363	0	16,244.0	3063	105,123	0
Storage1	657.5	625	701	0	3449.2	3315	3562	0	30.5	30	33	0	198.3	191	203	0
Storage2	714.4	629	785	0	274.0	251	315	0	15.4	14	31	0	260.7	250	319	0
Storage3	8285.5	5754	14,735	0	4690.8	4566	4897	0	164.8	162	212	0	12,110.7	1382	59,539	0
Storage4	132,241.7	98,016	234,082	0	1785.9	1748	1910	0	64.8	45	112	0	8528.5	3688	29,716	0
Storage5	477,195.8	294,830	878,911	0	6756.1	6472	6972	0	2586.0	2513	2746	0	99,613.3	11,728	426,232	0
SystemMgmt	866.8	761	950	0	1272.7	1224	1384	0	27.1	14	62	0	307.0	251	327	0
Telecom	2517.2	2379	2752	0	1693.3	1599	1764	0	19.4	14	33	0	1011.5	382	2692	0
CA_1	270.2	212	362	0	332.5	312	462	0	16.8	14	31	0	220.7	212	263	0
CA_2	1956.4	1767	2083	0	326.8	312	417	0	14.9	14	19	0	846.8	413	2314	0
CA_3	11,192.5	7767	16,862	0	360.7	312	417	0	23.4	14	62	0	7600.3	1162	36,896	0
CA_4	176.2	162	218	0	321.7	312	366	0	15.2	14	31	0	115.3	112	165	0
CA_5	311.4	262	366	0	337.2	312	417	0	14.6	14	16	0	264.6	212	515	0
CA_6	57,244.2	43,779	79,280	0	411.8	372	432	0	33.6	30	63	0	6211.4	1963	9093	0
CA_7	66,146.6	51,630	82,088	0	425.6	387	513	0	37.2	30	62	0	383,522.5	13,531	1,527,372	5
CA_8	NA	NA	NA	20	484.4	423	622	0	167.7	162	220	0	1,124,166.7	593,923	1,943,828	9
CA_9	20,290.0	13,029	34,596	0	338.6	312	412	0	32.3	30	62	0	16,228.9	1813	55,834	0
CA_10	748,894.9	426,985	1,122,601	0	360.4	312	393	0	64.0	62	80	0	212,706.6	18,923	690,574	2
CA_11	NA	NA	NA	20	432.2	412	513	0	167.1	162	214	0	394,353.4	46,915	1,456,322	4
VCA_1	7237.5	5555	9251	0	330.7	312	412	0	32.8	30	62	0	NA	NA	NA	NA
VCA_2	19,028.5	13,453	29,399	0	352.1	312	368	0	320.0	288	368	0	NA	NA	NA	NA
VCA_3	51,908.0	41,259	67,861	0	364.2	362	376	0	111.8	87	121	0	NA	NA	NA	NA
VCA_4	918,107.5	565,412	1,591,116	1	NA	NA	NA	20	379.1	313	417	0	NA	NA	NA	NA

translated into forms such as CNF formulas that can be processed by computers. Besides, some explicit constraints specified by the user may imply implicit constraints. For example, if we have two constraints "$p_1 == 1 \rightarrow p_2 == 1$" and "$p_1 == 1 \rightarrow p_3 == 1$", then combination $\{(p_1, 1), (p_3, 2)\}$ will never appear in any valid test case. Implicit constraints are specially considered in some algorithms.

There are plenty of mathematical construction techniques for covering array generation. These techniques are usually based on several construction rules. To the best of our knowledge, these construction rules are usually elaborately designed and can hardly be adapted to handle constraints. For example, the method of Hartman and Raskin (2004) uses mathematical construction for constructing arrays. When dealing with constraints, it just deletes invalid

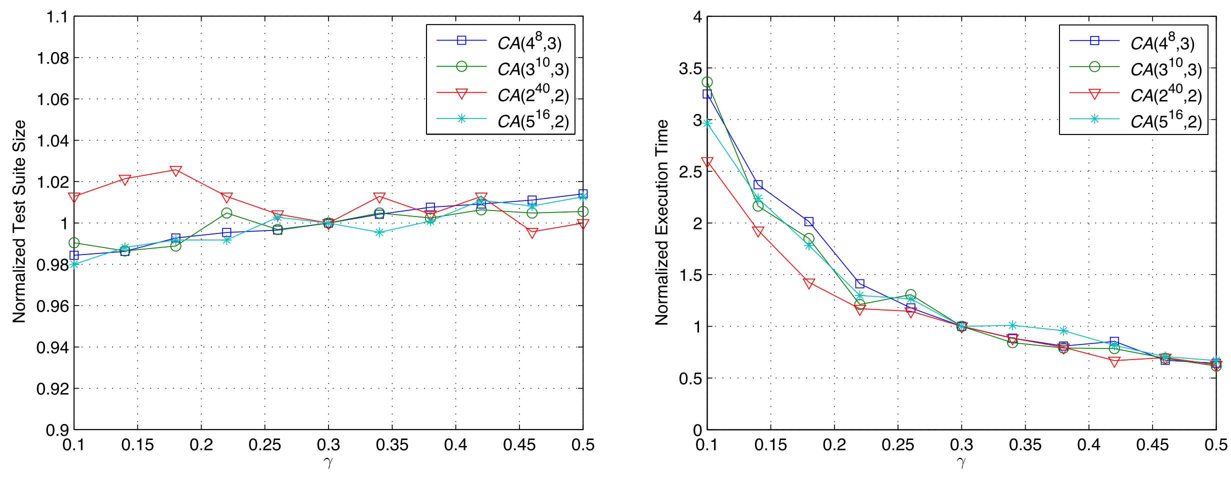

Fig. 9. Influence of γ on the test suite size and execution time.

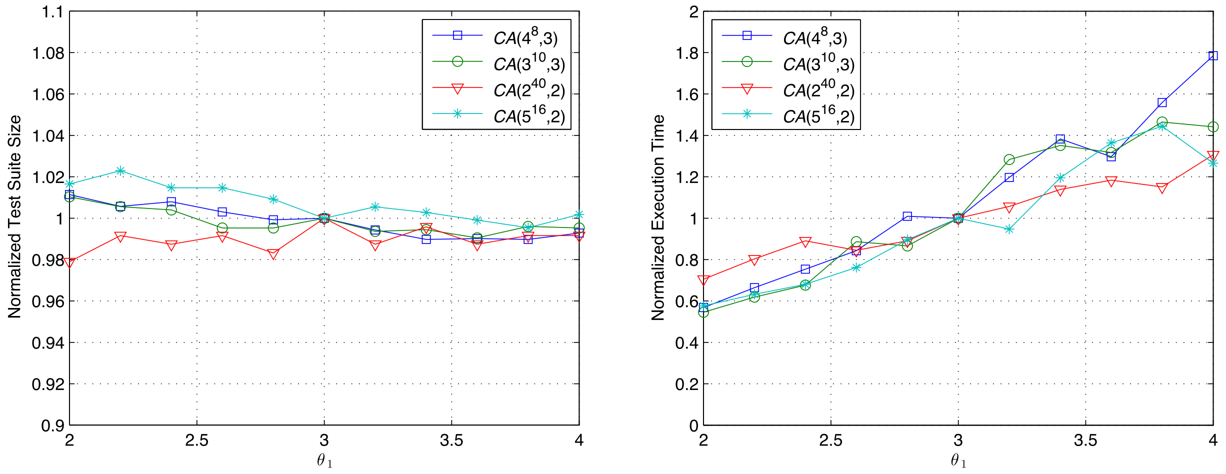

Fig. 10. Influence of θ_1 on the test suite size and execution time.

test cases. As a consequence coverage holes may occur, and the coverage requirement cannot be guaranteed.

Compared to mathematical construction techniques, computational CT test generation techniques are more flexible for constraint-handling.

Firstly, we talk about one-test-at-a-time algorithms. Algorithms based on this strategy generate the covering arrays by iteratively adding new test cases, until all target combinations are covered. To reduce the size of the covering array, the goal for generating each new test case is to maximize the number of newly covered target combinations. The main difference among these approaches is that they use different methods to assign values to parameters when generating the new test case. The Constrained Array Test System (CATS) (Sherwood, 1994) transforms the SUT model with constraints into several models without constraints, and expands all possible test cases from each model to yield the set of all valid test cases. Then it uses a greedy strategy to reorder the test cases. Each time it selects the test case that minimizes the number of uncovered combinations. The AETG algorithm (Cohen et al., 1997) generates each new test case by adding several candidate test cases. The candidate test cases are generated by greedily assigning parameter values in random order, and choosing the one having the greatest number of newly covered target combinations. They proposed two methods to handle constraints. The first method is similar to CATS. And the second method is to check the validity of each candidate test case after it is generated, and discard all invalid candidate test cases (Cohen et al., 1996). If all candidate test cases are invalid, AETG will exhaustively search for a valid test case. A drawback of this method is that when the constraints are complex and a large proportion of possible test cases are invalid, many candidate test cases will be invalid, so the chosen candidate test case may not cover enough uncovered target combinations. TCG (Tung and Aldiwan, 2000) and mAETG-SAT (Cohen et al., 2007a,b, 2008) are two AETG-like algorithms. When generating each candidate test case, the algorithm checks the validity of the partial test case after each assignment, and if the new assignment makes the resulting partial test case invalid, it undoes the assignment and tries another one. This ensures that there always exists at least one assignment for the unassigned parameters that completes the partial test case into a valid test case. PICT (Czerwonka, 2006) internally computes the set of all target combinations that are invalid, and also computes the set of all implicit forbidden combinations. Then during the test generation process, PICT avoids covering any explicit/implicit forbidden combinations, thus guaranteeing the partial assignment is valid and can be extended into a valid test case. The deterministic density algorithm (DDA) (Colbourn et al., 2004; Bryce and Colbourn, 2006, 2007a, 2009) is similar with AETG. The major differences are: (1) DDA generates only one test case for each row instead of a number of candidate test cases and (2) parameter assignments are based on *density*, which are computed according to the set of uncovered target combinations. And constraints can be supported by giving negative weights on forbidden combinations,

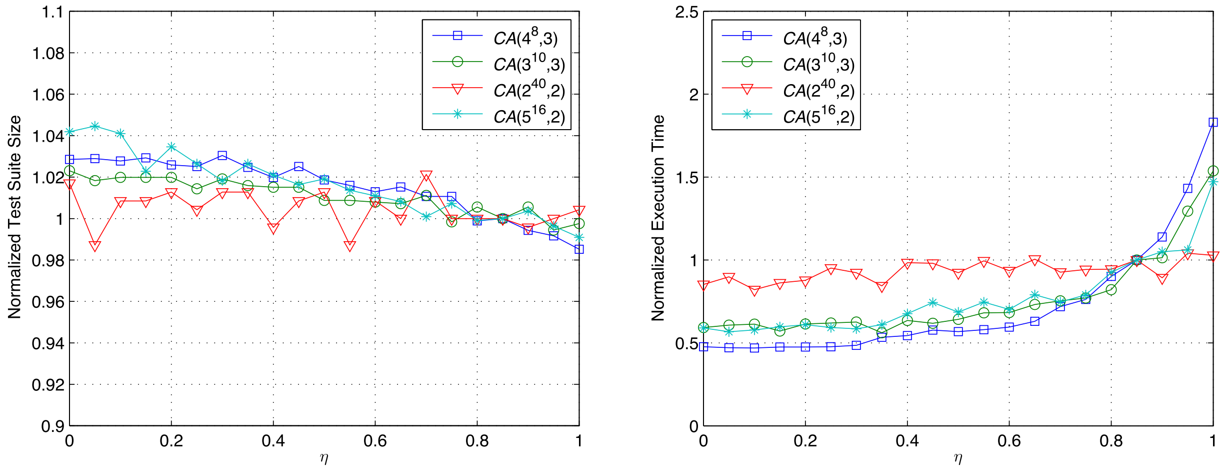

Fig. 11. Influence of η on the test suite size and execution time.

or by modifying the density formulae to emphasize more on target combinations. However this constraint-handling method only supports *soft constraints* which are constraints to avoid if possible. Calvagna and Gargantini (2008) generates each new test case by greedily selecting a set of consistent uncovered target combinations to assign some of the parameters, and using a model checker to find values for the remaining parameters. Validity check is performed using a model checker before each uncovered target combination is selected.

Secondly, we talk about the IPO-based algorithms (Lei et al., 2008). Algorithms of this type starts from a small covering array, and continuously extends the array into a larger array. Each time they extend the covering array by adding one column (parameter) to the array to cover as many newly covered target combinations as possible (the horizontal growth stage), and then modifying existing rows or adding additional rows to cover the remaining uncovered target combinations of the parameters related to the larger array (the vertical growth stage). Yu et al. (2013) proposed a modified algorithm IPOG-C to support constraints. The algorithm checks the validity of the partial test case whenever an assignment is performed. They also provided several optimizations to reduce the cost for validity check.

Thirdly, we talk about algorithms based on the whole-array-search strategy. These algorithms try to generate a covering array of a given size, which means the number of rows and columns is fixed, and the algorithms need to find an assignment to each entry to meet the coverage requirement. Hnich et al. (2005, 2006) translated the properties of a covering array into SAT constraints, and use a SAT solver to find a feasible solution. They mentioned constraints could be easily encoded for processing for SAT, however they did not give the details. Besides, their approach may not be scalable for large instances. Garvin et al. (2009, 2011) used simulated annealing to maximize the number of covered target combinations of the array. If all valid target combinations are covered, the array is a covering array. Their algorithm continuously modifies the current array to try to achieve a higher coverage. The algorithm performs validity check after each modification, and if the resulting array violates the constraints, it undoes the modification and tries another one.

Note that many approaches mentioned above are based on optimization techniques (for maximizing the number of covered target combinations). In these approaches, parameter assignment and constraint handling are often separated. The upper algorithm assigns values to parameters, but it does not know whether the assignment violates any constraint. So the algorithm checks the validity of the assignment, and if it is invalid, it undoes the assignment and tries another one. Unlike these approaches, our algorithm translates the problem of generating each new test case into a PBO problem, thus parameter assignment and constraint handling are integrated. The optimization solver performs assignments with backtracking strategy, which makes it possible to find better solutions.

Besides pseudo-Boolean optimization technique which is used in our work, many other optimization techniques are applied for CT test generation. These include meta-heuristic search technique such as simulated annealing, hill climbing, tabu search, genetic algorithm and ant colony algorithm. These techniques are usually used together with the one-test-at-a-time strategy (Shiba et al., 2004; Bryce and Colbourn, 2007b; Nie et al., 2012; Avila-George et al., 2013) or the whole-array-search strategy (Stardom, 2001; Ghazi and Ahmed, 2003; Cohen et al., 2003a,b; Garvin et al., 2009, 2011; Torres-Jimenez and Rodriguez-Tello, 2012). Similar to our algorithm, some of the one-test-at-a-time algorithms using optimization techniques also need to consider deciding when to stop the searching process. For example, in the AETG-GA algorithm (Shiba et al., 2004) which is based on genetic algorithm, the termination condition is when the total number of generated candidate tests exceeds a given number *M*. However, the input problem size may vary, and there is no universal *M* suitable for all problem instances. And letting the user choose an appropriate value for *M* is not a good idea. Our solver stopping mechanism can be easily adapted to these algorithms to solve the stopping problem.

In our approach, we translated our problem into a PBO problem and use an optimization solver to find a solution. Constraint solving techniques have also been used in other CT test generation techniques. Some techniques we mentioned earlier use a constraint solver to check the validity of test cases (including partial test cases) and arrays. Some other techniques translates the properties of a covering array or a test case into a *constraint satisfaction problem* (Hnich et al., 2005, 2006; Lopez-Escogido et al., 2008), an integer programming problem (Williams and Probert, 2002), or a model checking problem (Calvagna and Gargantini, 2008) and use a constraint solver or a model checker to find a feasible solution. However, the performance of these translation approaches degrades significantly as the problem size increases. EXACT (Yan and Zhang, 2008) uses backtrack search to find covering arrays. It assigns values to array cells in a backtracking way. When an assignment is performed, it uses a SAT solver to find value assignments implied by the current assignment, and to check the consistency of the current assignment. Besides, several heuristics and symmetry breaking techniques are applied to reduce the search space.

6. Conclusion

In real application scenarios of combinatorial testing, there are usually constraints in the SUT models. Ignoring these constraints will lead to invalid test cases and coverage holes. According to our evaluation, existing CT test generation algorithms with constraint-handling support perform worse when the number of constraints gets larger. In this paper, we propose a new CT test generation technique based on the one-test-at-a-time strategy, which translates the generation of each new test case into a linear pseudo-Boolean optimization problem. We have found that finding the optimal solution for each test case is very expensive and cannot efficiently reduce the test suite size. Experimental results show that a possible balance point is to keep the approximation ratio of each test case between 0.8 and 0.9. Since the solver we use does not support stopping when an approximation ratio is reached, we design a self-adaptive stopping mechanism to decide when to stop the solver when the solution is good enough. The mechanism can also be used in search-based algorithms based on the one-row-at-a-time strategy, which also suffer from the lack of a criterion to stop the searching process at an appropriate time. Experimental results show that our algorithm works fine with existing CT test generation benchmarks, and the constraint-processing ability is better than existing approaches when the number of constraints is large. Besides, we introduced a new method to handle shielding parameter constraints, which is a very common type of constraints but only a few works have addressed. All the methods described in this paper are implemented in our tool Cascade (Zhao et al., 2013), which can be found on our project website.[4]

However, there are still some points for possible improvement: (1) as we mentioned in Section 4.2, our stopping mechanism is based on the assumption that the translated PBO problem of the *i*th and the $(i+1)$th test case are similar. But there are some cases where the optimal value of the series of PBO problems drops very fast when the number of generated test cases increases. The current stopping mechanism may not work well in this situation, and we need to add some prediction methods to alleviate this

[4] http://lcs.ios.ac.cn/~zj/ct.html.

problem. Another work around is to use an optimization solver that supports stopping when a given approximation ratio is reached. (2) When the number of parameters, parameter levels and covering strength increase, the scale of the translated PBO problem increases exponentially. So our algorithm may work slower than greedy algorithms when the SUT model is large. Another reason why our approach is not very fast is that the solver we use is a general optimization solver. We observed that sometimes the optimization is really that hard to solve, and our stopping mechanism has chosen a good time to stop the solver. Possible speed enhancement may be achieved by adding heuristic knowledge specific to the CT test generation problem into the solving process.

Acknowledgement

This work is partially supported by the 973 Program of China (grant No. 2014CB340701) and the National Science Foundation of China (grant No. 61100064).

References

Avila-George, H., Torres-Jimenez, J., Gonzalez-Hernandez, L., Hernandez, V., 2013. Metaheuristic approach for constructing functional test-suites. IET Softw. 7 (2), 104–117.

Bryce, R.C., Colbourn, C.J., 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Inf. Softw. Technol. 48 (10), 960–970.

Bryce, R.C., Colbourn, C.J., 2007a. The density algorithm for pairwise interaction testing. Softw. Test. Verif. Reliab. 17 (3), 159–182.

Bryce, R.C., Colbourn, C.J., 2007b. One-test-at-a-time heuristic search for interaction test suites. In: Proc. of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), pp. 1082–1089.

Bryce, R.C., Colbourn, C.J., 2009. A density-based greedy algorithm for higher strength covering arrays. Softw. Test. Verif. Reliab. 19 (1), 37–53.

Calvagna, A., Gargantini, A., 2008. A logic-based approach to combinatorial testing with constraints. In: Proc. of the Second International Conference on Tests and Proofs (TAP'08), pp. 66–83.

Chen, B., Yan, J., Zhang, J., 2010. Combinatorial testing with shielding parameters. In: Proc. of the 17th Asia Pacific Software Engineering Conference (APSEC'10), pp. 280–289.

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1996. Method and system for automatically generating efficient test cases for systems having interacting elements. U.S. Patent 5,542,043 (issued 30.07.96).

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1997. The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23 (7), 437–444.

Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J., 2003a. Constructing test suites for interaction testing. In: Proc. of the 25th International Conference on Software Engineering (ICSE'03), pp. 38–48.

Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J., Collofello, J.S., 2003b. A variable strength interaction testing of components. In: Proc. of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03), pp. 413–418.

Cohen, M.B., Dwyer, M.B., Shi, J., 2007a. Exploiting constraint solving history to construct interaction test suites. In: Proc. of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007 (TAICPART-MUTATION'07), pp. 121–132.

Cohen, M.B., Dwyer, M.B., Shi, J., 2007b. Interaction testing of highly-configurable systems in the presence of constraints. In: Proc. of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07), pp. 129–139.

Cohen, M.B., Dwyer, M.B., Shi, J., 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. IEEE Trans. Softw. Eng. 34 (5), 633–650.

Colbourn, C.J., Cohen, M.B., Turban, R., 2004. A deterministic density algorithm for pairwise interaction coverage. In: Proc. of the 2004 IASTED International Conference on Software Engineering (SE'04), pp. 345–352.

Czerwonka, J., 2006. Pairwise testing in the real world. In: Proc. of the 24th Pacific Northwest Software Quality Conference (PNSQC'06), pp. 419–430.

Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2009. An improved meta-heuristic search for constrained interaction testing. In: Proc. of the 1st International Symposium on Search Based Software Engineering (SBSE'09), pp. 13–22.

Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. 16 (1), 61–102.

Gebser, M., Kaufmann, B., Schaub, T., 2012. Conflict-driven answer set solving: from theory to practice. Artif. Intell. 187–188, 52–89.

Ghandehari, L.S.G., Bourazjany, M.N., Lei, Y., Kacker, R.N., Kuhn, D.R., 2013. Applying combinatorial testing to the Siemens suite. In: Proc. of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST'13), pp. 362–371.

Ghazi, S.A., Ahmed, M.A., 2003. Pair-wise test coverage using genetic algorithms. In: Proc. of the 2003 Congress on Evolutionary Computation (CEC'03), pp. 1420–1424.

Hartman, A., Raskin, L., 2004. Problems and algorithms for covering arrays. Discret Math. 284 (1), 149–156.

Hnich, B., Prestwich, S., Selensky, E., 2005. Constraint-based approaches to the covering test problem. Recent Adv. Constraints, 172–186.

Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M., 2006. Constraint models for the covering test problem. Constraints 11 (2/3), 199–219.

Kuhn, D.R., Michael, J.R., 2002. An investigation of the applicability of design of experiments to software testing. In: Proc. of the 27th Annual NASA Goddard/IEEE, Software Engineering Workshop.

Kuhn, D.R., Kacker, R.N., Lei, Y., 2013. Introduction to Combinatorial Testing. Chapman and Hall/CRC, London, UK.

Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test. Verif. Reliab. 18 (3), 125–148.

Lopez-Escogido, D., Torres-Jimenez, J., Rodriguez-Tello, E., Rangel-Valdez, N., 2008. Strength two covering arrays construction using a SAT representation. MICAI 2008: Adv. Artif. Intell., 44–53.

Nie, C., Leung, H., 2011. A survey of combinatorial testing. ACM Comput. Surv. (CSUR) 43 (2), 11.

Nie, C., Wu, H., Liang, Y., Leung, H., Kuo, F.C., Li, Z., 2012. Search based combinatorial testing. In: Proc. of the 19th Asia-Pacific Software Engineering Conference (APSEC'12), pp. 778–783.

Segall, I., Tzoref-Brill, R., Farchi, E., 2011. Using binary decision diagrams for combinatorial test design. In: Proc. of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), pp. 254–264.

Sherwood, G., 1994. Effective testing of factor combinations. In: Proc. of the 3rd International Conference on Software Testing, Analysis, and Review (STAR'94).

Shiba, T., Tsuchiya, T., Kikuno, T., 2004. Using artificial life techniques to generate test cases for combinatorial testing. In: Proc. of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), pp. 72–77.

Stardom, J., 2001. Metaheuristics and the Search for Covering and Packing Arrays (Masters thesis). Simon Fraser University.

Tatsumi, K., Watanabe, S., Takeuchi, Y., Shimokawa, H., 1987. Conceptual support for test case design. In: Proc. of the 11th Annual International Computer Software and Applications Conference (COMPSAC'87).

Torres-Jimenez, J., Rodriguez-Tello, E., 2012. New bounds for binary covering arrays using simulated annealing. Inform. Sci. 185 (1), 137–152.

Tung, Y.W., Aldiwan, W.S., 2000. Automating test case generation for the new generation mission software system. In: Proc. of the IEEE Aerospace Conference, vol. 1, pp. 431–437.

Williams, A.W., Probert, R.L., 2002. Formulation of the interaction test coverage problem as an integer program. Test. Commun. Syst. XIV, 283–298.

Yan, J., Zhang, J., 2008. A backtracking search tool for constructing combinatorial test suites. J. Syst. Softw. 81 (10), 1681–1693.

Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R.N., Kuhn, D.R., 2013. An efficient algorithm for constraint handling in combinatorial test generation. In: Proc. of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST'13), pp. 242–251.

Zhao, Y., Zhang, Z., Yan, J., Zhang, J., 2013. Cascade: a test generation tool for combinatorial testing. In: Proc. of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13), pp. 267–270.

Zhiqiang Zhang received his bachelor's degree in Computer Science at University of Sciences and Technology of China in 2010. He is now a Ph.D. student at the Institute of Software, Chinese Academy of Sciences (ISCAS). His research interests mainly focus on software testing and fault localization.

Jun Yan received his bachelor's degree in E.E. from University of Science and Technology of China in 2001 and his Ph.D. degree in computer science from Institute of Software, Chinese Academy of Sciences (ISCAS) in 2007. He is now an associate research professor at ISCAS. His research interests include program analysis and software testing, constraint processing.

Yong Zhao is a Ph.D. student at the University of Colorado at Colorado Springs. He received his B.S. and M.S. degrees in Computer Science from East China Jiao Tong University and Institute of Software Chinese Academy Sciences respectively. Now his research interests include compiler construction, program analysis and optimization for high-performance computing.

Jian Zhang is a research professor at the Institute of Software, Chinese Academy of Sciences (ISCAS). His main research interests include automated reasoning, constraint satisfaction, source code analysis and software testing. He has served on the program committee of about 60 international conferences. He also serves on the editorial boards of several journals including Frontiers of Computer Science, Journal of Computer Science and Technology. He is a senior member of ACM, the IEEE, and China Computer Federation (CCF).