Cascade: a Test Generation Tool for Combinatorial Testing

Yong Zhao*, Zhiqiang Zhang*, Jun Yan†, Jian Zhang*
*State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
{zhaoy, zhangzq, zj}@ios.ac.cn
†Technology Center of Software Engineering
Institute of Software, Chinese Academy of Sciences, Beijing, China
yanjun@otcaix.iscas.ac.cn

***Abstract*—Combinatorial Testing (CT) is a black-box testing technique, which is used to test parameterized systems. Real applications often have some special testing requirements, especially parameter constraints. Our experimental results show that existing CT test generators cannot deal with constraints very well as the number of constraints increases. This is due to their constraint solving mechanisms, where constraint solving and parameter assignment are separated. In this paper, we propose a new CT test generation method based on constrained optimization, which combines these two together to mitigate the effect brought by the growing complexity of constraints. Our approach is implemented in a tool called *Cascade*, which can deal with common CT usage scenarios.**

***Keywords*-combinatorial testing; test generation; constrained optimization; parameter constraints;**

I. Introduction

Combinatorial Testing (CT) is an efficient black-box testing method, which is used to test parameterized systems. Since it greatly saves the test cost and provides very high fault coverage, this method has been paid more and more attention in industrial testing activities.

The first step for applying CT is to build a parameterized black-box model for the software under test (SUT). We need to identify the input parameters (or factors/components) affecting the system's behavior, and their constraints.

Now suppose the SUT has k parameters, each of which has s possible values. Exhaustive testing will require s^k test cases. This number will grow very large when s and k increase. CT can reduce the cost of testing while maintaining high fault detection ability. It is motivated by the observation that a significant number of failures are caused by the combination effect of a small number of parameters in many applications [12]. This type of faults are called *interaction faults*. If we can generate a test suite covering all combinations of every t parameters, then the test suite will be able to detect t-way interaction faults, i.e. at least one test case will fail when the SUT contains any interaction faults of t parameters. We usually use *covering arrays* as the test suite, which covers all combinations of every t parameters, meeting this demand. (Here t is called the *strength* of the covering array.)

Here we give an example scenario for applying CT:

This work is supported in part by the National Natural Science Foundation of China (under grant Nos. 60903049, 61061130541 and 91118007).

TABLE I
Online PC Retailing System Example

CPU	MB	HD	Mem	Graphics	Shipping
CPU1	MB1	500GB	2GB DDR2	NONE	$0.00
CPU2	MB2	1TB	4GB DDR2	Graphics1	$15.00
CPU3	MB3		4GB DDR3	Graphics2	
CPU4	MB4		8GB DDR3		

Example I.1. Table I shows a model for a PC retailing website. The user wants to buy a PC. He chooses preferred components, and submits an order to the website. The website should determine whether the selected components are compatible, check whether the order satisfies some promotion conditions, and give a total price. The next step, which is not considered in this model, is to let the user confirm the order, and proceed to the payment process. We can see the model has 6 parameters: CPU, motherboard (MB), hard disk (HD), memory (Mem), graphics card and shipping cost, each having 2 to 4 possible values. The parameter constraints are informally described as follows:

- Each component has a price.
- CPU and memory standards must be compatible with the motherboard.
- If the motherboard has an integrated graphics card, the user does not need to buy a dedicated graphics card if he has no demand for high performance.
- If the total price is greater than or equal to $300, or an item on promotion is chosen, the shipping will be free.

As CT is now being more and more used in industrial applications, there is a growing demand of the practitioners that CT become more user-friendly. This requires the CT test generator to possess the following two properties:

- It should be able to deal with common testing requirements.
- It should provide good usability, so that users can describe their problems in a clear and straightforward way.

However, according to our investigation, existing CT test generation tools, such as AETG [3], ACTS [13] and PICT [7], do not fully possess the above two properties, especially with regard to parameter constraints. The constraints in practical CT applications could be very complex and hard to express. Due to the limitations of existing tools, the user has to pay a lot of focus on how to translate the model into a test generator input,

978-0-7695-4993-4/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSTW.2013.37

rather than on building a high-quality model. For example, writing the input for Example I.1 for these tools will be difficult (see Section IV). On the other hand, these tools' constraint solving mechanisms make the time for generating a test suite increase rapidly with the growth of the constraint complexity (which will be shown in Section V of this paper).

This paper presents our work on CT test generation, which aims at providing these two properties. A new CT test generation method based on constrained optimization is introduced, which combines parameter assignments with constraint solving to improve the constraint processing ability. Our method outperforms existing tools when the number of constraints grow into very large scales. This method is implemented as a tool called *Cascade*, which supports common CT testing requirements, and it is highly user-friendly.

The rest of this paper is organized as follows: We introduce some common CT testing requirements in Section II. Then we give a brief overview of our new test generation method in Section III. And then we introduce *Cascade* in Section IV. Experimental results are shown in Section V. Related works are discussed in Section VI. And finally, we give the conclusions in Section VII.

II. Common CT Test Requirements

In this section, we give an introduction to typical CT test requirements. These are extensions to the classical CT concepts as described in Section I. These concepts include the following but not limited to them:

A. Mixed Covering Arrays

Letting all the parameters have the same value domains is no longer practical any more. Almost all practical SUT models have parameters of different value domains. This extension is called *Mixed Covering Array* (MCA).

B. Seeding

Sometimes there are some critical parameter combinations that must be covered. Seeding [3] allows the user to explicitly specify some combinations as *seeds* to let them covered by the covering array.

C. Variable Strength

There are situations where some parameters interact more than with other parameters. Specifying a higher universal covering strength upon all parameters is not a good choice, since unimportant parameter combinations will be covered, which will increase the number of test cases. Cohen et al. [4] introduced *variable strength*, which enables the user to specify different covering strengths upon different subsets of parameters.

D. Constraints

Dealing with *constraints*, as considered by both CT researchers and practitioners, is a very important aspect in CT test generation. Many SUT models have constraints, more or less, simple or complicated. And all the generated test cases should satisfy the constraints. Neglecting these constraints will cause many invalid test cases to be generated and hence some combinations will not be covered by valid test cases. This phenomenon is called *coverage holes*, which should not appear. CT test generation tools should take constraint solving into consideration.

III. Test Generation Using Constrained Optimization

In the last section, we introduced some common extensions to classical CT. Now we introduce our algorithm which can deal with all the four requirements mentioned above.

The algorithm is based on the one-row-at-a-time strategy. The top level is shown in Algorithm 1.

Algorithm 1 Kernel algorithm of *Cascade*

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

The one-row-at-a-time strategy first initializes the set of combinations needed to be covered according to the coverage requirement and the seeds specified by the user. In the while-loop, it iteratively generates a new test case covering as many uncovered combinations as possible. Then the algorithm removes all the combinations covered by the new test case from $combination\_set$ (line 7). The loop terminates if no more combinations can be covered (line 9). After the loop terminates, all the remaining combinations in $combination\_set$ are not satisfiable.

Generating a new test case in the while-loop requires the new test case to cover as many uncovered combinations as possible, which is an optimization problem. In order to deal with constraints, most existing algorithms, such as [3], [7], [13], use some greedy methods or heuristics to assign values to parameters to generate a partial or complete test case. They may call an external solver or other methods to determine whether this test case (or partial test case) satisfies the constraints, or to determine which values can be assigned to a parameter. This generate-and-test mechanism is inefficient when constraints grow complex, which is proven by our experimental results in Section V.

In contrast, our algorithm encodes the problem of generating a new test case into a constrained optimization problem (line 3) (a constraint is added to ensure the new test case covers at least an uncovered combination), then it uses an external solver to generate the new test case (line 4). This integrates optimization and constraint solving together. The test-and-retry process is done inside the solver, but not in the upper algorithm level. Since constraint solvers are good at solving constraints, this will save a lot of resources on constraint processing.

In our approach, we use a *pseudo-Boolean optimization* (PBO) solver called *clasp*[1]. The PBO problem is essentially identical to the *0-1 integer programming* problem. The pseudo-Boolean (PB) variables are binary and can take values in $\{0, 1\}$, and the solver constraints are in the form of linear constraints. The PBO problem also have a linear objective function, which is to be minimized over the PB variables.

IV. Cascade: A New CT Test Generation Tool

Algorithm 1 has set the base of our work. We have implemented this algorithm as a complete tool named *Cascade*. It has a very user-friendly input language. The grammar of constraints is very close to the C programming language, which we believe is more straightforward than existing tools. Please see Section VI for detailed comparison. Figure 1 shows the general framework of *Cascade*.

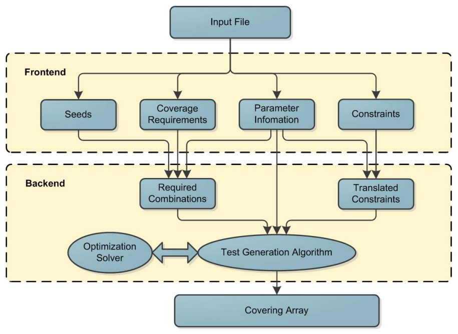

Fig. 1. General Framework of *Cascade*

Cascade provides a feature called *auxiliary parameters*, which was first introduced in [14]. They are very useful in writing complex constraints. *Cascade* treats auxiliary parameters as regular parameters, except that they are not in any coverage requirement by default. These parameters are translated into solver variables. *Cascade* also allows the user to use *type casts* from/to boolean, integer and double types.

Now let's come back to Example I.1. Part of the input file for *Cascade* is shown in Figure 2. (See [16] for the complete input file.) We can see the input file for *Cascade* is clean and straightforward, while writing this model for existing tools will be challenging and non-intuitive, since they don't have auxiliary parameters, and do not support arithmetic operators on floating-point parameters (see Table II).

V. Experimental Results

In this section, we provide some preliminary results of the performance of *Cascade* on processing complex constraints in comparison with several existing tools. All the experimental subjects can be found at [16].

1) Random Constraints: We randomly generate constraints in the form of *forbidden combinations*. Random constraints are very difficult since they are highly unstructured. We vary the number of constraints to compare the time of each test generator. The results are shown in Figure 3.

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

```
[PARAMETERS]
string CPU: "CPU1", "CPU2", "CPU3", "CPU4";
string MB: "MB1", "MB2", "MB3", "MB4";
string Mem: "2GB DDR2", "4GB DDR2",
            "4GB DDR3", "8GB DDR3";
string HD: "500GB", "1TB";
string Graphics: "NONE", "Graphics1", "Graphics2";
double Shipping: 0.00, 15.00;

aux double CPUPrice: 42.00, 65.00, 34.00, 49.00;
...
aux double GraphicsPrice: 0.00, 127.00, 54.00;
aux string CPUSocket: "LGA 775", "AM3";
aux string DDRVersion: "DDR2", "DDR3";
aux bool HighPrice: true, false;

[STRENGTHS]
default: 2;
CPU, MB, Mem: 3; // higher strength on critical components

[CONSTRAINTS]
// item prices
CPU=="CPU1" -> CPUPrice==42.00;
...
Graphics=="Graphics2" -> GraphicsPrice==54.00;
// component compatibility
(CPU=="CPU1" || CPU=="CPU2") -> CPUSocket=="LGA 775";
...
(Mem=="2GB DDR2" || Mem=="4GB DDR2")->DDRVersion=="DDR2";
...
MB=="MB1"->(CPUSocket=="LGA 775" && DDRVersion=="DDR2");
...
(MB=="MB2" || MB=="MB3") -> Graphics != "NONE";
// shipping cost
CPUPrice + ... + GraphicsPrice >= 300.00
    <-> HighPrice == true;
(HighPrice || CPU == "CPU2" || MB == "MB2")
    <-> Shipping == 0.00;
```

Fig. 2. Input Model File of Example I.1 for *Cascade*

2) Structured Constraints: For structured constraints, we fix the number of integer parameters, and forbid all the combinations of a subset of parameters, whose sum is greater than a given number. The results are shown in Figure 4.

In Figure 3 and Figure 4, the missing points of ACTS and PICT mean that the tool crashes or the running time exceeds a timeout of 30 minutes. The numbers of test cases generated by the three tools are very close for all the instances, so we do not show them in this paper.

From both the running time of random and structured experiments, we can see that the time growth of *Cascade* is much slower compared with the other two. However, the time for generating test suites for models with a small number of constraints is relatively long, but it is still acceptable.

VI. Related Works

Test generation has long been a hot topic in CT. Popular CT test generation algorithms include algorithms based on the one-row-at-a-time strategy [3], and the IPO-family algorithms [13]. Hnich et al. [10], [11] encoded the problem into SAT constraints to generate covering arrays, but their approach does not support constraints.

There are some CT test generation approaches considering constraint handling. Due to the space limit, only part of them are discussed here. AETG [3] requires remodeling the input space to support constraints. The DDA algorithm [1] supports soft constraints (parameter combinations not desirable to be

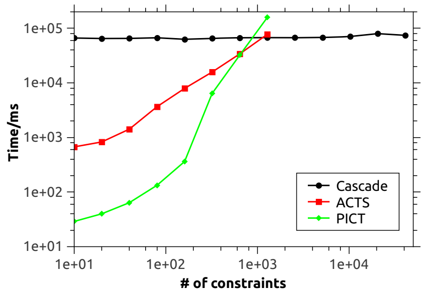

Fig. 3. Experimental Results On Random Constraints

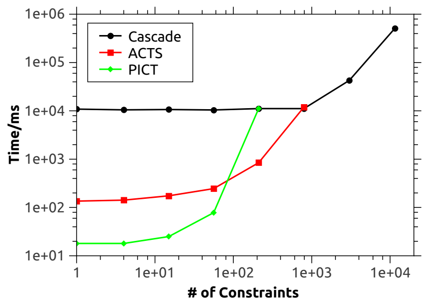

Fig. 4. Experimental Results On Structured Constraints

covered), but it does not support hard constraints (which must be strictly satisfied). Cohen et al. [5], [6] proposed several strategies which adapt the AETG algorithms to handle constraints. Calvagna and Gargantini [2] uses a model checker to generate feasible test cases. Garvin et al. [8], [9] used simulated annealing to search for a small covering array, while using a SAT solver to check constraint satisfaction. What all the above approaches have in common is that they handle constraints in two layers: On the upper layer, they assign values to parameters to cover as more new parameter combinations as possible; On the lower layer, they use constraint solvers to determine constraint satisfaction. Different from these approaches, *Cascade* encodes the whole problem of generating each test case into a constrained optimization problem. The constraint handling and coverage optimization are all handled by the solver.

A comparison of the features of *Cascade* with a selection of available tools is shown in Table II. ("P_Types" = parameter types, "A_Optrs" = arithmetic operators, "R_Optrs" = relational operators, "B_Optrs" = Boolean operators). We can see *Cascade* can deal with nearly all situations other tools can do.

VII. Conclusions and Ongoing Work

This paper presents an overview of our work on CT test generation. We proposed a new algorithm which combines parameter assignment with constraint solving using constrained optimization. The constraint solving mechanism increases the ability of constraint processing.

TABLE II
Test Generation Feature Comparison

	AETG [3]	PICT [7]	ACTS [13]	Cascade
MCA	✓	✓	✓	✓
Seeding	✓	✓	✗	✓
VCA	✓	✓	✓	✓
P_Types	string numerical	string numerical	string integer boolean	string integer double boolean
A_Optrs	✗	✗	$+, -, \times$ $/, \%$	$+, -, \times$ $/, \%$
R_Optrs	$\in, \notin, \neq, >$ $\geq, <, \leq$	$\in, =, \neq, >$ $\geq, <, \leq$ wildcard match	$=, \neq, >$ $\geq, <, \leq$	$=, \neq, >$ $\geq, <, \leq$
B_Optrs	$\wedge, \vee, \rightarrow$	$\neg, \wedge, \vee, \rightarrow$	$\neg, \wedge, \vee, \rightarrow$	$\neg, \wedge, \vee$ $\rightarrow, \leftrightarrow, \oplus$
Type Cast	✗	✗	✗	✓

We have developed the prototype tool *Cascade*. It supports many advanced features, and it is easy to use. A demo version of *Cascade* is now available at [15]. We are still working hard on its development. The future works include: 1) Extending the input language to support more types of constraints; 2) Refining the encoding of the test generation problem into a constrained optimization problem; 3) Increasing the solving speed for generating test suite for models with a small number of constraints.

References

[1] R.C. Bryce and C.J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10): 960–970, 2006.
[2] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. *Tests and Proofs*, 66–83, 2008.
[3] D. Cohen, S. Dalal, M. Fredman and G. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. on Software Engineering*, 23(7): 437–443, 1997.
[4] M.B. Cohen and C.J. Colbourn. Variable strength interaction testing of components. *Proc. COMPSAC'03*, 413–418.
[5] M.B. Cohen, M.B. Dwyer, J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. *Proc. ISSTA'07*, 129–139.
[6] 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. on Software Engineering*, 34(5): 633–650, 2008.
[7] J. Czerwonka. Pairwise testing in the real world. *Proc. PNSQC'06*, 419–430.
[8] B.J. Garvin, M.B. Cohen and M.B. Dwyer. An improved meta-heuristic search for constrained interaction testing. *Proc. SSBSE'09*, 13–22.
[9] B.J. Garvin, M.B. Cohen and M.B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1): 61–102, 2011.
[10] B. Hnich, S. Prestwich and E. Selensky. Constraint-based approaches to the covering test problem. *Recent Advances in Constraints*, 172–186, 2005.
[11] B. Hnich, S.D. Prestwich, E. Selensky and B.M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2): 199-219, 2006.
[12] D.R. Kuhn and M.J. Reily. An investigation of the applicability of design of experiments to software testing. *Proc. SEW'02*, 91-95.
[13] Y. Lei, R. Kacker, D. Kuhn, V. Okun and J. Lawrence. IPOG/IPOD: Efficient Test Generation for Multi-Way Software Testing. *Journal of Software Testing, Verification and Reliability*, 18(3): 125–148, 2008.
[14] I. Segall, R. Tzoref-Brill and A. Zlotnick. Simplified modeling of combinatorial test spaces. *Proc. ICST'12*, 573–579.
[15] Z. Zhang et al. Combinatorial testing toolkit, http://lcs.ios.ac.cn/~zhangzq/ct-toolkit.html.
[16] Z. Zhang and Y. Zhao. Model files used in this paper, http://lcs.ios.ac.cn/~zhangzq/cascade-subjects.zip.