Faulty Interaction Identification via Constraint Solving and Optimization*

Jian Zhang, Feifei Ma, and Zhiqiang Zhang

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences

Abstract. Combinatorial testing (CT) is an important black-box testing method. In CT, the behavior of the system under test (SUT) is affected by several parameters/components. Then CT generates a combinatorial test suite. After the user executes a test suite and starts debugging, some test cases fail and some pass. From the perspective of a black box, the failures are caused by interaction of several parameters. It will be helpful if we can identify a small set of interacting parameters that caused the failures. This paper proposes a new automatic approach to identifying faulty interactions. It uses (pseudo-Boolean) constraint solving and optimization techniques to analyze the execution results of the combinatorial test suite. Experimental results show that the method is quite efficient and it can find faulty combinatorial interactions quickly. They also shed some light on the relation between the size of test suite and the ability of fault localization.

1 Introduction

For many complex software systems, there are usually different components (or options or parameters) which interact with each other. Combinatorial testing (CT) [1,7,9] is an important black-box testing technique for such systems. It is a generic technique and can be applied to different testing levels such as unit testing, integration testing and system testing.

Using some automatic tools, we can generate a reasonable test suite (e.g., a covering array) which achieves a certain kind of coverage. For instance, pair-wise testing covers all different pairs of values for any two parameters.

After the system is tested, we get the execution results. Typically some test cases pass, while a few others fail. We would like to identify the cause of the failure. In CT, we adopt a parametric model of the system under test (SUT). We assume that the system fails due to the interaction of some parameters/components. Such failure-causing parameter interactions are called *faulty combinatorial interactions* (FCIs).

There are already some works on identifying FCIs, e.g., [10,12,2,8,11,14,5]. This paper proposes a new approach which is based on constraint solving and

* Supported in part by the National Natural Science Foundation of China under grants No. 61070039 and No. 61100064.

A. Cimatti and R. Sebastiani (Eds.): SAT 2012, LNCS 7317, pp. 186–199, 2012.

optimization techniques. Our approach is a test result analysis technique. The test results of the combinatorial test suite are used as input, and no additional test cases are generated. One benefit of our approach is that it can identify all possible solutions for the combinatorial test suite. Here, a solution is a set of suspicious FCIs, such that once the SUT has exactly these FCIs, the execution of the combinatorial test suite will be the same as the real execution result. Also, as we can find all possible solutions, the number of possible solutions can be used as a criterion to measure the precision of the solutions. Specifically, if the number of possible solutions is large, then each possible solution has a low precision. Our approach also provides evidence of the tradeoff between reducing the size of the test suite and enhancing its fault localization ability. (Most traditional CT techniques aim at generating small test suites to reduce the cost of testing. They provide good fault detection, but do not support fault localization very well.)

The remainder of this paper is organized as follows: Section 2 gives some definitions. Section 3 introduces our approach, in which we formulate the problem as a constraint satisfaction problem (CSP). Section 4 discusses translating the problem into a pseudo-Boolean optimization problem. Section 5 presents some experimental results. Section 6 compares our approach with some related work. Section 7 gives the conclusions and some implication of our results for CT.

2 Definitions

Now we give some formal definitions related to CT.

Definition 1. *(SUT model) A model* $SUT(k, s)$ *has* k *parameters* $p_1, p_2, \ldots, p_k$. *The vector* s *is of length* k, *i.e.* $\langle s_1, s_2, \ldots, s_k \rangle$, *where* s_i *indicates the number of possible values of the parameter* p_i. *The domain of* p_i *is* $D_i = \{1, 2, \ldots, s_i\}$. *And* s_i *is called the* level *of* p_i.

Definition 2. *(Test case) A test case* t *is a vector of length* k, *representing an assignment to each parameter with a specific value within its domain. The execution result of a test case is either* `pass` *or* `fail`.

Definition 3. *(Test suite) A test suite* T *is a set of test cases* $\{t_1, t_2, \ldots, t_m\}$.

Definition 4. *(CI) A combinatorial interaction (CI) is a vector of length* k, *which assigns* t *parameters to specific values, leaving the rest* $k - t$ *parameter values undetermined (undetermined values are denoted by '-'). Here* t *is the size of the CI.*

A CI represents the interaction of the t parameters with assigned values. The undetermined parameters ('-') are just placeholders, and do not participate in the interaction.

Definition 5. *(CI containment) A CI* P_1 *is contained by another CI* P_2 *if and only if all determined parameters of* P_1 *are determined parameters of* P_2, *and these parameters are assigned the same values in* P_1 *and* P_2. *A CI* P *is contained by a test case* t *if and only if all determined parameters in* P *have the same values as those in* t.

Definition 6. *(FCI) A faulty combinatorial interaction (FCI) is a CI such that all possible test cases containing it will fail.*

Note that if a CI is an FCI, then all CIs containing it are FCIs. Suppose we have an FCI P_1, and P_2 contains P_1. Then all test cases containing P_2 will contain P_1, and all of these test cases will fail. Thus P_2 is an FCI. For example, suppose `(1,-,2,-)` is an FCI, then `(1,1,2,-)` and `(1,-,2,3)` are also FCIs. In this paper, we only identify the minimal FCIs, i.e. FCIs containing no other FCIs.

Example 1. Table 1 shows a covering array for pair-wise testing of an online payment system. There are 9 test cases, and for two of them the SUT fails. The braces ("{}") indicate the value combinations causing the failure of the SUT, which we do not know in advance. The test results are shown in the last column. We can see that this covering array can detect all FCIs of size 2, but we cannot tell what they are just from the test results. (e.g. the failure of the 3rd test case may also be caused by the interaction of Firefox and Apache.)

Table 1. A Sample Covering Array

Client	Web Server	Payment	Database	Exec
Firefox	WebSphere	MasterCard	DB/2	`pass`
Firefox	.NET	UnionPay	Oracle	`pass`
Firefox	{Apache}	{Visa}	Access	`fail`
IE	WebSphere	UnionPay	Access	`pass`
{IE}	Apache	MasterCard	{Oracle}	`fail`
IE	.NET	Visa	DB/2	`pass`
Opera	WebSphere	Visa	Oracle	`pass`
Opera	.NET	MasterCard	Access	`pass`
Opera	Apache	UnionPay	DB/2	`pass`

3 Formulation as a Constraint Satisfaction Problem

Suppose the system under test (SUT) has k attributes or parameters or components. The i^{th} attribute may take one value from the following set: $D_i = \{1, 2, \ldots\}$

Assume that there are already m test cases. Some of them failed, while the others passed. The CT fault model assumes that failures are caused by value combinations (i.e. several parameters are assigned to specific values). We would like to know which combinations in the failing test cases caused the failure (e.g., made the SUT crash).

For simplicity, we first assume that there is only one failing test case in the test suite. We would like to identify the value combination (i.e., FCI) in this test case. It can be represented by a vector like this: $\langle x_1, x_2, \ldots, x_k \rangle$, where each x_i can be 0 or a value in the set D_i. When $x_j = 0$, it means that the j'th attribute does not appear in the FCI.

Our way to solve the FCI identification problem is to formulate it as a constraint satisfaction problem (CSP) or satisfiability (SAT) problem. In a CSP, there are some variables, each of which can take values from a certain domain; and there are also some constraints. Solving a CSP means finding a suitable value (in the domain) for each variable, such that all the constraints hold.

We have already given the "variables" in the CSP. Now we describe the "constraints". Roughly, a failing test case should match the FCI vector; and a passing test case should not match it. More formally, for a passing test case $(v_1, v_2, \ldots, v_k)$, we need a constraint like this:

```
    ( NOT(x1=0) AND NOT(x1=v1) )
 OR ( NOT(x2=0) AND NOT(x2=v2) )
 OR ......
 OR ( NOT(xk=0) AND NOT(xk=vk) )
```

For a failing test case $(w_1, w_2, \ldots, w_k)$, we need the following constraint:

```
    ( (x1=0) OR (x1=w1) )
AND ( (x2=0) OR (x2=w2) )
AND ......
AND ( (xk=0) OR (xk=wk) )
```

In this way, we obtain the "constraints" in the CSP. After solving the CSP, we get the values of the variables x_i $(1 \leq i \leq k)$. Then, deleting the 0's, we get the FCI.

Example 1. (Continued) Assume that the values in the table can be represented by integers as follows:

- `Client` can be: 1. `Firefox`; 2. `IE`; 3. `Opera`.
- `WebServer` can be: 1. `WebSphere`; 2. `.NET`; 3. `Apache`.
- `Payment` can be: 1. `MasterCard`; 2. `UnionPay`; 3. `VISA`.
- `Database` can be: 1. `DB/2`; 2. `Oracle`; 3. `Access`.

Suppose that the FCI is denoted by the vector $\langle x_1, x_2, x_3, x_4 \rangle$. The domain of each variable x_i is [0..3]. We can derive the following constraints from the first 3 test cases:

```
    ( NOT(x1=0) AND NOT(x1=1) )
 OR ( NOT(x2=0) AND NOT(x2=1) )
 OR ( NOT(x3=0) AND NOT(x3=1) )
 OR ( NOT(x4=0) AND NOT(x4=1) ) ;

    ( NOT(x1=0) AND NOT(x1=1) )
 OR ( NOT(x2=0) AND NOT(x2=2) )
 OR ( NOT(x3=0) AND NOT(x3=2) )
 OR ( NOT(x4=0) AND NOT(x4=2) ) ;
```

```
    ( (x1=0) OR (x1=1) )
AND ( (x2=0) OR (x2=3) )
AND ( (x3=0) OR (x3=3) )
AND ( (x4=0) OR (x4=3) ) .
```

More constraints can be derived from the other test cases, which are omitted here.

More than One FCIs

Example 1. (Continued) Now suppose that there are two different FCIs, denoted by $\langle x_1, x_2, x_3, x_4\rangle$ and $\langle y_1, y_2, y_3, y_4\rangle$, respectively. From the first test case (which passed), we obtain the following constraint:

```
(   ( NOT(x1=0) AND NOT(x1=1) )
 OR ( NOT(x2=0) AND NOT(x2=1) )
 OR ( NOT(x3=0) AND NOT(x3=1) )
 OR ( NOT(x4=0) AND NOT(x4=1) ))
AND
(   ( NOT(y1=0) AND NOT(y1=1) )
 OR ( NOT(y2=0) AND NOT(y2=1) )
 OR ( NOT(y3=0) AND NOT(y3=1) )
 OR ( NOT(y4=0) AND NOT(y4=1) )).
```

From the third test case (which failed), we obtain the following constraint:

```
(   ( (x1=0) OR (x1=1) )
AND ( (x2=0) OR (x2=3) )
AND ( (x3=0) OR (x3=3) )
AND ( (x4=0) OR (x4=3) ))
OR
(   ( (y1=0) OR (y1=1) )
AND ( (y2=0) OR (y2=3) )
AND ( (y3=0) OR (y3=3) )
AND ( (y4=0) OR (y4=3) )).
```

Symmetry Breaking

For the above example with two FCIs, suppose by solving the constraints we get $\langle x_1 = v_1, x_2 = v_2, x_3 = v_3, x_4 = v_4\rangle$ and $\langle y_1 = v'_1, y_2 = v'_2, y_3 = v'_3, y_4 = v'_4\rangle$. If we exchange the respective values of the two vectors, we get $\langle x_1 = v'_1, x_2 = v'_2, x_3 = v'_3, x_4 = v'_4\rangle$ and $\langle y_1 = v_1, y_2 = v_2, y_3 = v_3, y_4 = v_4\rangle$. Obviously it is another solution to the constraints. However, the two solutions represent the same set of FCIs. They are symmetric solutions.

A symmetry is a one to one mapping (bijection) on decision variables that preserves solutions and non-solutions. Symmetries can generate redundant search space, so it is very important to eliminate symmetries while solving the problem.

For a system with n FCIs, there are $n!$ permutations of the variable vectors of FCIs in the aforementioned constraints, hence results in as many symmetries. To break such symmetries, we introduce lexicographic order on the variable vectors. In the above example, we add the following constraints which ensure that one vector is lexicographically smaller than the other.

```
    ( (x1<=y1) )
AND ( (x1<y1) OR (x2<=y2) )
AND ( (x1<y1) OR (x2<y2) OR (x3<=y3) )
AND ( (x1<y1) OR (x2<y2) OR (x3<y3) OR (x4<y4) )
```

Apparently all symmetries caused by permutations of FCIs can be eliminated with the above constraints.

Determining the Number of FCIs

To use our method, one should specify the number of FCIs in advance. But we do not know the actual number. So we gradually increase the number. We start from $n = 1$; if the problem is unsatisfiable, then n is increased by 1; ... We repeat this procedure until the problem becomes satisfiable.

4 Translation to Pseudo-Boolean Optimization Problem

By checking the satisfiability of the aforementioned constraints, one or a set of FCIs can be discovered. However, there might be many solutions to the constraint satisfaction problem, and it is desirable to find the optimal one according to some criterion. A reasonable objective is to minimize the size of FCI, i.e., to maximize the number of '0's, so that the FCI is the most general one. As a result, in this paper we are investigating FCI identification as an optimization problem instead of a decision problem.

This problem can be naturally formulated as a Pseudo-Boolean Optimization (PBO) problem. In its broadest sense, a pseudo-Boolean function is a function that maps Boolean values to a real number.

A linear pseudo-Boolean constraint has the following form

$$\Sigma_i a_i x_i \geq b$$

where $x_i \in \{0, 1\}$ is a Boolean variable and a_i, b are integers.

A pseudo-Boolean constraint is nonlinear if it contains the product of Boolean variables. Such a constraint has the following form

$$\Sigma_i a_i (\Pi_k x_{i,k}) \geq b$$

A pseudo-Boolean optimization problem is to maximize (minimize) a pseudo-Boolean expression subject to a set of pseudo-Boolean constraints.

4.1 Encoding

Suppose there are w failing cases for the SUT, and we are looking for n ($n \leq w$) FCIs. We use Boolean variable $P_{i,j,v}$ to indicate that the j^{th} ($1 \leq j \leq k$) parameter of the i^{th} ($1 \leq i \leq n$) FCI $x_{i,j}$ takes value v, or formally, $P_{i,j,v} \equiv (x_{i,j} = v)$. When there are more than one FCIs, we also introduce auxiliary Boolean variable $E_{t,i}$ to indicate that the t^{th} ($1 \leq t \leq w$) failing case is caused by the i^{th} FCI. The P-variables are called *primary variables*, and the E-variables are called *auxiliary variables*.

Assume that our goal is to maximize the total number of zero values of all FCIs, we have the following objective function:

$Minimize \quad -\Sigma_i \Sigma_j P_{i,j,0}$

There are four types of pseudo-Boolean constraints:

1. Basic constraints: Constraints that guarantee the validity of the encoding. We have to make sure that each parameter of each FCI can take only one value. So we add constraints $\Sigma_v P_{i,j,v} = 1$ for all $1 \leq i \leq n, 1 \leq j \leq k$. Here the variable v ranges over $D_i \cup \{0\}$.
2. Constraints for passing cases: To facilitate encoding, we make slight changes to the original constraints by removing the innermost `AND` operator. For instance, in *Example 1* we replace `(NOT(x1=0) AND NOT(x1=1))` with `(x1=2 OR x1=3)`, and `(NOT(x2=0) AND NOT(x2=1))` with `(x2=2 OR x2=3)`, and so on. Generally, for each passing case $V = \{v_1, v_2, \ldots, v_k\}$, we add:

$$\bigwedge_{1 \leq i \leq n} \Sigma_{j=1}^{k} \Sigma_{v, v \neq 0, v \neq v_j} P_{i,j,v} \geq 1$$

3. Constraints for failing cases: If there is only one FCI, we simply translate the original constraints into pseudo-Boolean constraints. For each failing case $V = \{v_1, v_2, \ldots, v_k\}$, we add:

$$\bigwedge_{1 \leq j \leq k} P_{1,j,0} + P_{1,j,v_j} = 1$$

If the number of target FCIs n is more than 1, each failing case must match at least one FCI, which results in the constraints

$$\Sigma_{i=1}^{n} E_{t,i} \geq 1$$

for all $1 \leq t \leq w$.

In addition, for the t^{th} failing case $V = \{v_1, v_2, \ldots, v_k\}$, we have

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq k} -E_{t,i} + P_{i,j,0} + P_{i,j,v_j} \geq 0$$

And

$$\bigwedge_{1 \leq i \leq n} E_{t,i} + \Sigma_{1 \leq j \leq k} \Sigma_{v, v \neq 0, v \neq v_j} P_{i,j,v} \geq 1$$

which means if the t^{th} failing case matches the i^{th} FCI, then the j^{th} parameter of the i^{th} FCI either takes value 0 or takes value v_j, and vice versa.

Besides, we must make sure that there is no useless FCI. Each FCI must match at least one failing case. So for all $1 \leq i \leq n$, we add:

$$\Sigma_{t=1}^{w} E_{t,i} \geq 1$$

4. Symmetry breaking constraints: A direct way to encode the inequality (e.g. `x1<y1`) is to enumerate all assignments allowed by the inequality and assert that at least one assignment is true. Therefore, the constraints that state the i^{th} FCI is lexicographically smaller than the $(i+1)^{\text{th}}$ FCI are encoded as follows:

$$\begin{array}{l} (\bigwedge_{1 \leq j \leq k-1} \Sigma_{l=1}^{j} \Sigma_{v_1<v_2} P_{i,l,v1} P_{i+1,l,v2} + \Sigma_v P_{i,j,v} P_{i+1,j,v} \geq 1) \\ \bigwedge \quad \Sigma_{l=1}^{k} \Sigma_{v_1<v_2} P_{i,l,v1} P_{i+1,l,v2} \geq 1 \end{array}$$

Unlike the other three classes of PB constraints, these PB constraints are nonlinear.

4.2 Tool Support

There are quite some tools for solving pseudo-Boolean constraints. In our work, we use the tool `clasp` [4], a conflict-driven nogood learning answer set solver, to solve the optimization problem. `clasp` can be applied as an ASP solver, as a SAT solver, or as a PB solver. We have developed a prototype tool which translates the original problem into the input of `clasp`.

5 Experiments

All the experiments in this paper are performed on a laptop with Intel CPU: Core i5 M540, 2.53GHz, running Ubuntu 11.10.

5.1 Simulation Results

In this part, we present some experimental simulation results. The results are shown in Table 2. For each row, an experiment is performed as follows:

1. We give an SUT model, as well as a set of FCIs in the SUT. Suppose the set of FCIs is unknown before our tool is used.
2. A combinatorial test suite is generated according to the SUT model and a given coverage strength t.
3. Label the test suite results with the set of FCIs. Any test case containing one or more FCIs will be labeled as `fail`; while other test cases are labeled as `pass`. This step simulates the testing procedure for real systems.
4. We apply our tool to the test results to generate possible solutions. Then we analyze the results.

Table 2. Results for Locating FCIs on Test Suites

Exp #	Test Suite	nTC	nFCI	sFCI	nFT	nSol/nSol_NSB	tSB (s)	tNSB (s)
1	$CAM(3^4, 2)$	9	1	1(1/1)	3	1/1	0.001	=
2				2(1/2)	1	6/6	0.001	=
3			2	1(2/1)	5	1/6	0.001	0.001
4				2(2/2)	2	36/72	0.002	0.003
5	$CA(3^4, 2)$	13	1	1(1/1)	5	1/1	0.001	=
6				2(1/2)	2	1/1	0.001	=
7			2	1(2/1)	7	2/36	0.001	0.002
8				2(1/2)	2	1/1	0.001	=
9	$CA(2^8, 2)$	7	1	1(1/1)	4	1/1	0.001	=
10				2(1/2)	1	13/13	0.001	=
11				3(1/2)	1	6/6	0.001	=
12			2	1(2/1)	6	4/24	0.002	0.002
13				2(2/1.5)	3	12/48	0.003	0.004
14				1&2(2/1)	5	3/90	0.002	0.005
15	$CA(2^{20}, 2)$	10	1	1(1/1)	6	1/1	0.001	=
16				2(1/2)	3	4/4	0.001	=
17				3(1/2)	2	9/9	0.001	=
18			2	1(2/1)	7	4/216	0.006	0.022
19				2(1/2)	4	1/1	0.001	=
20	$CA(3^{20}, 2)$	24	1	1(1/1)	9	1/1	0.001	=
21				2(1/2)	3	1/1	0.001	=
22				3(1/2)	1	3/3	0.003	=
23			2	1(2/1)	16	1/18	0.017	0.007
24				2(2/2)	5	5/10	0.015	0.008
25				3(1/2)	2	3/3	0.001	=
26	$TS1(2^8)$	10	1	3(1/3)	6	1/1	0.001	=
27	$TS2(2^8)$	18	2	1&2(2/1.5)	14	1/54	0.002	0.004
28	$CA(6^{10}, 3)$	526	4	1-4(3/2)	97	1/162	0.177	0.157
29			8	1-4(5/1.8)	180	6/-	7.177	NA
30	$CA(3^{50}, 3)$	133	2	2&3(2/2.5)	20	1/6	0.137	0.052
31			4	1-4(3/2)	56	30/-	1.854	NA
32	$CA(3^{50}, 4)$	579	2	2&3(2/2.5)	83	1/192	0.302	0.279
33			4	1-4(4/2.5)	256	1/-	53.185	NA
34	$CA(3^{100}, 4)$	169	2	2&3(2/2.5)	22	1/6	0.920	0.201
35			4	1-4(3/2)	74	1/39366	11.975	51.781

In Table 2, column "nTC" shows the number of test cases, "nFCI" shows the number of FCIs we give in advance, column "sFCI" shows the size of FCIs we give in advance (the "a/b" in the parenthesis shows the number and average size of identified FCIs), column "nFT" shows the number of failing test cases, column "nSol / nSol_NSB" shows the number of solutions found when symmetry breaking is used or not used, columns "tSB" and "tNSB" show the running times when symmetry breaking is used and not used, respectively. An "=" in the tNSB column means that when the number of FCIs is 1, there is actually no symmetry breaking constraint, so there is no significant difference between tSB and tNSB.

"NA" stands for "not available", which means that the solver did not find an optimal solution within 300 seconds.

$CA(s^k, t)$ represents a covering array of strength t, having k parameters of level s (i.e. each parameter has s possible values). A covering array of strength t covers all possible value combinations among t parameters. A $CAM(s^k, t)$ is a covering array having the minimum number of test cases among all $CA(s^k, t)$'s. In Table 2, the CAs are generated by PICT of Microsoft [3]; and CAM(3^4,2) is the covering array in Example 1. *TS1*(2^8) and *TS2*(2^8) are two test suites generated using the algorithms in [14]. Each of the two test suites tries to localize FCIs in one failing test case.

Our technique only analyzes the execution results of the test suite. Details about running the SUT are not considered in our evaluation. From Table 2, we can see the average time for finding each possible solution is less than 0.010s for most of the small cases (exp # 1-27). And our method scales up to some large cases (exp # 28-35).

Applying symmetry breaking sometimes works worse than not adopting symmetry breaking, since the additional nonlinear constraints slows the constraint solving. But for some large cases, symmetry breaking can greatly reduce the solving time.

We also have the following observations:

- The number of solutions may be greater than 1. The reason is that the test suite may not be sufficient to localize the faults. The number of possible solutions can be used to measure the fault localization ability of the test suite. The more solutions we get, the less accurate the FCI localization is.
- From the results of $CAM(3^4, 2)$ and $CA(3^4, 2)$, we can see the number of solutions decreases with increase of the number of test cases. This is because a larger number of test cases will provide more information about the faults.
- From the results of *TS1* and *TS2*, we can see that the test cases used by the methods in [14] are sufficient to localize the fault, and the number of solutions is 1 for each experiment.
- The results show that when the number and size of the FCIs of the SUT (which we declared in advance) grows, the number of solutions grows, or the number and size of identified FCIs are more likely to be inconsistent with that of the FCIs of the SUT. Both provide evidence that the increasing of the number and size of FCIs of the SUT will make fault localization more difficult. Also, when the number and size of FCIs is 1, the number of solution for $CA(s^k, 2)$ is always 1. These conclusions conform to that of [8].

5.2 Experiment on a Real System

We also applied our technique to a real system. The experiment subject is a module of the Traffic Collision Avoidance System (TCAS) benchmark.

First, we build a parameterized model for TCAS. Here we used the same SUT model as used in [6]. The model has some input parameters: 6 of the parameters are of level 2, 3 parameters are of level 3, 1 parameter is of level 4, and 2

parameters are of level 10. So there are $3 \times 2^5 \times 4 \times 10^2 \times 3 \times 2 \times 3 = 691200$ possible test cases in total.

The TCAS benchmark has 41 faulty versions and 1 correct version. We perform our experiment in the following steps:

1. Select a faulty version among the 41 versions.
2. Set a coverage strength t, and generate a covering array of strength t.
3. Execute the covering array on the selected faulty version. The execution result of a test case is determined by comparing the output of the faulty version with that of the correct version.
4. Apply our technique to the execution results of the covering array, in order to identify the FCIs.

In our evaluation, we conducted experiments on the first 15 versions of TCAS. The results are shown in Table 3. In the table, row "nTC" shows the number of test cases in the covering array. Row "nFTC" shows the number of failing test cases. Rows "nSol" and "nSol_NSB" show the number of solutions when symmetry breaking is used and not used, respectively. In these rows, "AP" means that all test cases passed and our technique is not applied. "NA" stands for "not available", i.e., the solver cannot reach an optimal solution within 300 seconds. Row "nFCI" shows the number of FCIs. Row "asFCI" shows the average size of identified FCIs.

In our experiments, the covering arrays of strength 5 and 6 are very large (4294 and 11333 test cases respectively). These large test suites are likely to trigger more FCIs, which makes it less possible to find a solution if we assume a small number of FCIs in advance. So we assume that the number of FCIs is large too. This will result in a large number of (pseudo-Boolean) variables and constraints, and make it difficult for the solver to find a solution. Indeed, most experiments for strength 5 and 6 got an "NA", so we do not show them in Table 3.

From the results, we can see that the performance of our technique is not so good for the TCAS example. The reason is that the size of FCIs is relatively large. The consequence is two-fold. On one hand, covering arrays of lower strengths are not likely to trigger SUT failures, and most experiments with lower strengths got an "AP". On the other hand, covering arrays of higher strengths will need more test cases, and more likely there are quite some FCIs. Both will make the number of constraints large, resulting an "NA".

6 Related Works

There are different kinds of FCI identification methods. Some adaptive approaches like [13,10,14] take one failing test case as input, then generate and execute additional test cases to identify the FCIs in the failing test case. Other adaptive approaches like [11,5] process the whole test suite. These methods proceed in an iterative way. In each iteration, they analyze the test suite to identify suspicious FCIs. Then they generate and execute some additional test cases to refine the suspicious FCIs.

Table 3. Experimental Results for TCAS

Version	v1			v2			v3		
t	2	3	4	2	3	4	2	3	4
nTC	100	402	1410	100	402	1410	100	402	1410
nFTC	0	0	0	0	0	0	0	3	1
nSol	AP	AP	AP	AP	AP	AP	AP	2	7
nSol_NSB	AP	AP	AP	AP	AP	AP	AP	4	7
nFCI	-	-	-	-	-	-	-	2	1
asFCI	-	-	-	-	-	-	-	4	4
tSB (s)	-	-	-	-	-	-	-	0.024	0.038
tNSB (s)	-	-	-	-	-	-	-	0.022	=

Version	v4			v5			v6		
t	2	3	4	2	3	4	2	3	4
nTC	100	402	1410	100	402	1410	100	402	1410
nFTC	0	0	2	0	6	21	0	6	9
nSol	AP	AP	1	AP	18	$\geq$10	AP	15	182
nSol_NSB	AP	AP	1	AP	432	$\geq$10	AP	90	7056
nFCI	-	-	1	-	4	-	-	3	4
asFCI	-	-	5	-	4	-	-	4	5
tSB (s)	-	-	0.030	-	1.014	-	-	0.072	1.256
tNSB (s)	-	-	=	-	5.213	-	-	0.113	6.836

Version	v7			v8			v9		
t	2	3	4	2	3	4	2	3	4
nTC	100	402	1410	100	402	1410	100	402	1410
nFTC	0	0	1	0	0	1	0	0	1
nSol	AP	AP	4	AP	AP	3	AP	AP	4
nSol_NSB	AP	AP	4	AP	AP	3	AP	AP	4
nFCI	-	-	1	-	-	1	-	-	1
asFCI	-	-	4	-	-	4	-	-	4
tSB (s)	-	-	0.032	-	-	0.032	-	-	0.032
tNSB (s)	-	-	=	-	-	=	-	-	=

Version	v10			v11			v12		
t	2	3	4	2	3	4	2	3	4
nTC	100	402	1410	100	402	1410	100	402	1410
nFTC	1	7	16	1	9	23	3	14	57
nSol	1	12	700	1	60	NA	1	NA	NA
nSol_NSB	1	72	NA	1	1440	NA	6	NA	NA
nFCI	1	3	7	1	4	$\geq$ 10	3	$\geq$ 8	$\geq$ 10
asFCI	2	5	5.57	2	4.75	-	2	-	-
tSB (s)	0.002	0.045	14.035	0.002	0.227	-	0.244	-	-
tNSB (s)	=	0.049	NA	=	0.865	-	0.257	-	-

Version	v13			v14			v15		
t	2	3	4	2	3	4	2	3	4
nTC	100	402	1410	100	402	1410	100	402	1410
nFTC	0	2	11	1	3	4	0	6	21
nSol	AP	5	34	1	12	30	AP	18	NA
nSol_NSB	AP	5	360	1	24	180	AP	432	NA
nFCI	-	1	3	1	2	3	-	4	$\geq$ 10
asFCI	-	5	5.67	2	3.5	4.67	-	4	-
tSB (s)	-	0.006	0.168	0.002	0.025	0.375	-	0.996	-
tNSB (s)	-	=	0.259	=	0.021	0.823	-	5.313	-

There are also some works aiming at generating covering arrays having fault localization abilities, such as LDA [2] and ELA [8]. One problem with these approaches is that the test suite is large and will cost a lot of resources during testing period. Besides, the generation of these covering arrays is still a problem.

Another kind of approaches aim at analyzing all the test execution results to identify the FCIs. Our technique belongs to this kind. The benefits of these methods are that only the test results are used as input, and no additional test

cases are generated. However, these methods suffer from insufficient test results, which provide insufficient information about FCIs and will lower the precision of FCI identification. Previous works of this kind include [12]. It uses classification tree analysis (CTA) on the test results. The FCIs are represented as a tree structure, and a path from the root to each failing leaf node corresponds to an FCI. (An example of classification trees is shown in Fig. 1. The p_i's represent the parameters, and the numbers on the branches are their values.) This approach is fast and insensitive to occasional failures. However, in some situations, only a very small set of test cases fail, which means the input data for CTA is highly unbalanced. Then CTA will have very bad performance when processing this kind of data. Another point is that all the FCIs should contain the same parameter on the root (it can be observed from Fig. 1), but this is not always the case. Compared with the CTA approach, our technique will not face the difficulties of CTA, and we can find all possible solutions of FCIs. Each of these solutions will lead to exactly the same test result as the input test results.

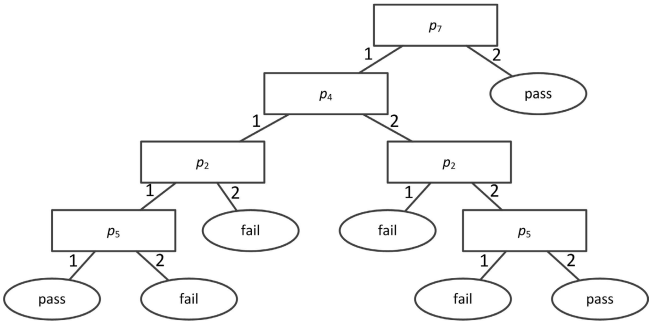

Fig. 1. An Example Classification Tree

7 Concluding Remarks

This paper proposes a new automated approach for identifying FCIs. It is based on pseudo-Boolean constraint solving and optimization techniques, and it is quite effective. The method has been implemented as a prototype tool. Preliminary results are encouraging. In most cases, it can provide a few FCIs which are helpful to the user when debugging. Of course, there are still some works to be done. For example, there can be different encodings of the problem, which are worth investigating.

Sometimes there may be a number of possible solutions for the generated constraints, which means that the localized FCIs may not be unique. This is because the test suite is insufficient to provide enough information about the failure. The more solutions we get, the less accurate the FCI localization is. One direction of our future works is to make a balance between reducing the size of the test suite and increasing the ability of fault localization.

Our work in this paper is another application of SAT/constraint solving (and optimization) techniques in software engineering. On the other hand, some benchmarks can be generated from such applications. They might be interesting and challenging to the SAT community.

References

1. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. IEEE Trans. on Softw. Eng. 23(7), 437–444 (1997)
2. Colbourn, C., McClary, D.: Locating and detecting arrays for interaction faults. J. of Combinatorial Optimization 15(1), 17–48 (2011)
3. Czerwonka, J.: Pairwise testing in realworld: Practical extensions to test case generator. In: Proc. PNSQC 2006, pp. 419–430 (2006)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proc. IJCAI 2007, pp. 386–392 (2007), `http://www.cs.uni-potsdam.de/clasp/`
5. Ghandehari, L.S.G., Lei, Y., Xie, T., Kuhn, D.R., Kacker, R.: Identifying failure-inducing combinations in a combinatorial test set. In: Proc. ICST 2012, pp. 370–379 (2012)
6. Kuhn, D.R., Okun, V.: Pseudo-exhaustive testing for software. In: Proc. SEW 2006, pp. 153–158 (2006)
7. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. on Softw. Eng. 30(6), 418–421 (2004)
8. Martínez, C., Moura, L., Panario, D., Stevens, B.: Algorithms to Locate Errors Using Covering Arrays. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 504–519. Springer, Heidelberg (2008)
9. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys 43(2) (2011)
10. Shi, L., Nie, C., Xu, B.: A Software Debugging Method Based on Pairwise Testing. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005, Part III. LNCS, vol. 3516, pp. 1088–1091. Springer, Heidelberg (2005)
11. Wang, Z., Xu, B., Chen, L., Xu, L.: Adaptive interaction fault location based on combinatorial testing. In: Proc. QSIC 2010, pp. 495–502 (2010)
12. Yilmaz, C., Cohen, M.B., Porter, A.A.: Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans. on Softw. Eng. 32(1), 20–34 (2006)
13. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. on Soft. Eng., 183–200 (2002)
14. Zhang, Z., Zhang, J.: Characterizing failure-causing parameter interactions by adaptive testing. In: Proc. ISSTA 2011, pp. 331–341 (2011)