Applying Combinatorial Testing to the Siemens Suite

Laleh Shikh Gholamhossein Ghandehari[1], Mehra N. Bourazjany[1], Yu Lei[1], Raghu N. Kacker[2], D. Richard Kuhn[2]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas 76019, USA
[2]Information Technology Laboratory National Institute of Standards and Technology, Gaithersburg, Maryland 20899, USA

***Abstract*- Combinatorial testing has attracted a lot of attention from both industry and academia. A number of reports suggest that combinatorial testing can be effective for practical applications. However, there are few systematic, controlled studies on the effectiveness of combinatorial testing. In particular, input parameter modeling is a key step in the combinatorial testing process. But most studies do not report the details of the modeling process. In this paper, we report an experiment that applies combinatorial testing to the Siemens suite. The Siemens suite has been used as a benchmark to evaluate the effectiveness of many testing techniques. Each program in the suite has a number of faulty versions. The effectiveness of combinatorial testing is measured in terms of the number of faulty versions that are detected. The experimental results show that combinatorial testing is effective in terms of detecting most of the faulty versions with a small number of tests. In addition, we report the details of our modeling process, which we hope to shed some lights on this critical, yet often ignored step, in the combinatorial testing process.**

Keywords- Combinatorial Testing, Input Modeling, Software Testing.

I. INTRODUCTION

Combinatorial testing has attracted a lot of attention from researchers. The key observation in combinatorial testing is that most software failures are caused by interactions of only a few input parameters. A t-way combinatorial test set is built to cover all the t-way interactions, where t is typically a small integer [10][5]. If test parameters and values are properly modeled, a t-way test set is able to expose all failures that involve no more than t parameters.

A number of empirical reports suggest that combinatorial testing can be effective for practical applications [1][2][6]. Most studies in these reports were designed to show that combinatorial testing could be applied to different types of applications. Thus, they were not controlled studies for evaluating the effectiveness of combinatorial testing. There are two notable exceptions. Kuhn et al. studied several fault databases and found that all the faults in these databases are caused by interaction of no more than six parameters [8][9]. These studies did not perform actual combinatorial testing on the subject systems. Schroeder et al. compared the effectiveness of t-way testing to random testing in a controlled study [13]. They selected two software applications used in their laboratory as subject programs, and manually seeded a number of faults to measure fault detection effectiveness.

In this paper, we report an experiment that applies combinatorial testing to the Siemens suite [16]. The Siemens suite has been used as a benchmark to evaluate the effectiveness of many testing techniques [2][6] [17]. Each program in the suite has a number of faulty versions. The effectiveness of combinatorial testing is measured in terms of the number of faulty versions that are detected. The results show that most of the faulty versions are detected by a small number of test cases. For example, all 32 faulty versions of *replace* program are detected by a 2-way test set containing only 192 tests. Furthermore, the results show that combinatorial testing is more effective than random testing.

We also report the details of our modeling process, which is a critical, yet often ignored step in the combinatorial testing process. Our approach consists of three main steps. First we create an abstract model for the system. This model consists of abstract parameters and values. On the one hand, abstraction reduces the modeling complexity that has to be managed at one time. On the other hand, abstraction helps to discover aspects that need to be tested. Second we generate a combinatorial test set based on the abstract model. Existing combinatorial test generation tools such as ACTS [1] can be used in this step. Third, we derive concrete tests from the abstract tests. These concrete tests are then used to perform the actual testing.

It is important to note that whereas the programs in the Siemens suite are relatively small, in terms of lines of code, and have a small number of input parameters, their input spaces are complex. For example, *replace* has 564 lines of code and 3 input parameters. However, its abstract model contains 20 abstract parameters and 36 constraints. The input parameters have different features and characteristics that must be considered for testing, e.g. one of the input parameters is a regular expression.

The remainder of this paper is organized as follows. In section II, we describe our approach for applying combinatorial testing. Section III reports experimental results that demonstrate the effectiveness of our modeling. Section IV discusses existing work on input space modeling. Section V provides concluding remarks.

II. APPROACH

In this section, we explain our approach to apply combinatorial testing. The approach consists of three major steps: (1) Create an abstract model, (2) Generate an abstract test set, and (3) Derive concrete tests. We use the

replace program in the Siemens suite, to explain each task in detail.

A. *Create abstract model*

This step has two major tasks: (1) define abstract parameters and values, (2) define relations and constraints.

1) Define abstract parameters and values

First, we analyze the system specification and identify factors that may affect the behavior of the system. These factors are candidates for abstract parameters. The equivalence partitioning approach is used to define the values of each abstract parameter.

We use the *replace* program in the Siemens suite to show how we define abstract parameters and values based on its specification. The *replace* program has three inputs, *pattern*, *substitute* and *input text*. The program finds every match of the *pattern* in the *input text* and replaces it with the *substitute*.

The *pattern* is a restricted form of regular expression. Table I shows the metacharacters that can be used in *pattern*. Note that the @ character can have different meanings, depending on the next character. If a character other than *n* and *t* appears after @, the program ignores it. For example, *@e* matches *e*. But when @ appears at the end of the *pattern*, the program behave as if it is a simple character and matches with @. For example, *e@* matches *e@*.

The *substitute* is a string that allows only three metacharacters to be used. These include two metacharacters, *@t* and *@n*, as shown in Table I, and a metacharacter &, which represents the string that matches the *pattern*. For example, if the string that matches the *pattern* is *ab* and the substitute is *a&c*, all *ab* strings in the file are replaced with *aabc*.

Table II shows the abstract model of the *replace* program for *pattern* and *substitute*. There are a total of 20 parameters in the model. The parameters with prefix *pat* are identified for pattern, and the parameters with prefix *sub* are identified for *substitute*. Note that these parameters are abstract as they are not the actual input parameters taken by the *replace* program.

The key modeling decision is twofold. First, each metacharacter is identified to be an abstract parameter. Our motivation is that the core logic of the *replace* program is dealing with these metacharacters. Thus, we consider each metacharacter to be an important factor that could affect the program behavior. Special attention is paid to metacharacters * and &. These two metacharacters can be combined with other meta or regular characters. An abstract parameter is identified for each possible combination. For example, *pat_question** represents the combination where a question mark appear before *.

Second, the values of each abstract parameter (i.e., metacharacter) are identified based two considerations. The first consideration is whether or not a parameter appears in the pattern (or substitute). Two values, *off* and *on*, can be used to represent the two cases. The second consideration is the following: If a parameter does appear in the pattern (or substitute), where does it appear? Thus,

TABLE I PATTERN'S METACHARACTER

Metacharacter	Description
?	Matches every character.
*	Matches the preceding pattern element zero or more times.
[-]	Matches a single character that is in the specified range. For example [a-c] matches “a”, “b” and “c”.
[^]	Matches every character except the ones inside brackets.
@t	Matches a tab.
@n	Matches the end of a line.
%	Matches the beginning of a line. (BOL)
$	Matches the end of a line. (EOL)

the *on* value identified earlier is further divided into three abstract values, *begin*, *middle*, and *end*. In Table II, all the parameters but four have four values, *off*, *begin*, *middle*, and *end*. The four exceptions, i.e., *pat_BOL*, *pat_EOL*, *pat_@n*, and *pat_@*, only have two values, *on* and *off*, because they can only appear in a particular position by nature. For example, BOL (i.e., %) by definition can only appear in the beginning of the pattern.

Now we discuss how to model the third input parameter, i.e., the *input text*, of the *replace* program. We consider that an *input text* consists of a sequence of lines. The key observation is that a line is relevant from the testing perspective only if it contains a match or mismatch of the *pattern*. Assume that the *pattern* consists of *k* elements. The *input text* is modeled such that it consists of k + 2 lines. The first line matches the *pattern*. The second line matches all the elements but the first in the *pattern*. The third line matches all the elements but the second in

TABLE II THE ABSTRACT MODEL OF REPLACE

Parameters	Values
pat_character[1]	[off, begin, middle, end]
pat_question[2]	[off, begin, middle, end]
pat_range[3]	[off, begin, middle, end]
pat_negate[4]	[off, begin, middle, end]
pat_@t	[off, begin, middle, end]
pat_@character	[off, begin, middle, end]
pat_question*	[off, begin, middle, end]
pat_character*	[off, begin, middle, end]
pat_range*	[off, begin, middle, end]
pat_negate*	[off, begin, middle, end]
pat_@t*	[off, begin, middle, end]
pat_@character*	[off, begin, middle, end]
pat_BOL[5]	[off,on]
pat_EOL[6]	[off,on]
pat_@n	[off,on]
pat_@	[off,on]
sub_character	[off, begin, middle, end]
sub_@n	[off, begin, middle, end]
sub_@character	[off, begin, middle, end]
sub_&	[off, begin, middle, end]

[1]Regular character
[2]? metacharacter
[3][-] metacharacter
[4][^] metacharacter
[5]% metacharacter
[6]$ metacharacter

the *pattern*, and so on. The last line does not match any element in the *pattern*. Note that we do not consider cases where a mismatch is due to multiple, but not all, of the elements in the *pattern*. This is essentially a trade-off made between test effort and test coverage.

2) Define relations and constraints

Relations are used to create parameter groups that can be covered at different strengths. Furthermore, parameters in different groups are independent and thus their combinations do not have to be tested. In our experiments, we used the default relation where all the parameters are considered to be in the same group. In retrospect, the parameters for *pattern* could be put into one group and the parameters for *substitute* in a second group. This would allow us to reduce the number of tests.

Constraints are used to exclude combinations that are not valid from the domain semantics. For the *replace* program, a total of 36 constraints are specified. All these 36 constraints are concerned with the position values of different parameters. In particular, in each test, there shall be only one parameter that has the value *begin* or *end*.

B. *Generate abstract tests*

In this step, an abstract test set is generated using an existing combinatorial test generation tool [10]. We used the ACTS tool [1]. ACTS can generate a combinatorial test set with strength 2 through 6. Note that these tests are abstract in that they cannot be directly executed. Instead, concrete tests must be derived first, which is discussed below.

C. *Derive concrete tests*

A scheme is needed to derive a concrete test from each abstract test. Conceptually, such a scheme consists of two parts. The first part is to map each abstract value to a concrete value. An abstract value is typically identified in a way such that it represents an equivalence group, i.e., a group of values that are equivalent to each other in terms of how they could affect the system behavior. Thus, it is sufficient to map an abstract value to any value in its equivalence group. For example, in the replace program, the abstract value, middle, represents all the positions those are neither at the beginning nor at the end. The specific position is often not important.

The second step is to map an abstract test to a concrete test. This part builds on the first step. In addition, it needs to map abstract parameters to concrete parameters. Recall that abstract parameters are identified to represent factors that could affect the system behavior. There typically does not exist a one-to-one mapping between abstract and concrete parameters. In fact, there are often more abstract parameters than concrete parameters. For example, for the replace program, there exist 20 abstract parameters, which need to be mapped to three concrete input parameters.

As an example, consider the abstract test in Figure 1(a) and the concrete test in Figure 1(b) for the *replace* program. In this example, the value of *pat_BOL* is *on*, so "%" is put at the beginning of the pattern. Similar, "@n"

a- Abstract test

Parameters	Values
pat_character	middle
pat_question	middle
pat_range	middle
pat_negate	middle
pat_@t	off
pat_@character	off
pat_question*	off
pat_character*	off
pat_range*	off
pat_negate*	off
pat_@t*	off
pat_@character*	off
pat_BOL	on
pat_EOL	off
pat_@n	on
pat_@	off
sub_character	begin
sub_@n	end
sub_@character	off
sub_&	middle

b. Concrete test

Parameters	Values
Pattern	%a?[a-e][^a]@n
Substitute	a&@n
Input file	1. abef 2. gabef 3. bef 4. aef 5. abf 6. abe 7. abefg 8. gbfag

Figure 1 An Example of Abstract Test and its Concrete Test

is placed at the end of the pattern. Other parameters, whose values are *middle*, are placed in the middle of the pattern. For *pat_character*, *pat_range* and *pat_negate* a, [a-e] and [^a] are put in pattern. Similarly, the *substitute* is created based on the corresponding parameter values in the abstract test.

The last row of Figure 1(b) shows different lines in the input file. The first line, *abef*, matches the pattern, since *a* matches with *a*, *b* matches with question mark, *e* matches with [*a-e*] element, and *f* matches with [*^a*]. Also, the first line matches *%* at the beginning and *@n* at the end.

Each line from line 2 to 7 matches all but one element in the *pattern*. For example the second line has the exact string *abef* which matches the pattern. However, since it is not at the beginning of the line (i.e., there is *g* at the beginning), the first element, *%*, in the pattern is not matched. The third line violates *a* in the pattern, and so on. The last line, i.e., line 8, does not match any element in the pattern.

Note that the scheme used to derive concrete tests from abstract tests is often specific to the subject application. However, such a scheme typically can be fully automated. This is the case for our experiments, where we wrote a program for each subject program to automate this process.

III. EXPERIMENT

We used the Siemens suite as our subject programs [16]. The Siemens suite contains 7 programs and each of these programs contains a number of faulty versions. The Siemens suite also provides an error-free version and a test pool for each program.

Table III represents properties of subject programs. The second column shows the number of lines of uncommented code. The third column shows the number

TABLE III SUBJECT PROGRAMS

Program	LOC	Procedures	#Faulty Versions
print_tokens	726	20	7
print_tokens2	570	21	10
replace	564	21	32
schedule	412	18	9
schedule2	374	16	10
tcas	173	8	41
totinfo	565	16	23

of procedures. The forth column shows the number of faulty versions for each program.

Two programs, *printtokens* and *printtokens2*, have the same specification but different implementations. Since the input space model is independent from the source code, these programs share the same model. Similarly, two programs *schedule* and *schedule2* have the same specification and thus share the same model. Therefore, in this section, we present five input models for the Siemens suite programs. Note that the input model for *tcas* is given in [8] and is included here for completeness.

In our experiments, we focus on interaction faults. As a result, our models are not designed for boundary testing or invalid testing. We believe most boundary and invalid faults are one-way faults, and they can be detected more efficiently using a different model where the focus is to identify special values of individual parameters. However, this belief needs to be validated by more experiments, which is beyond the scope of this paper.

Specifications of the programs are not provided by the benchmark. To understand what each program is supposed to do, we had to inspect the source code. (A search on the Internet did not find any such specification either.) To avoid potential bias in developing the model, only the source code of the error-free version was used. That is, we were not aware of the faults during the modeling process. Nonetheless, this is an internal threat to validity that needs to be considered.

We start with 2-way testing, and then move to 3-way testing, and so on, until (1) all faulty versions are detected; or (2) testing at the current strength does not detect any faulty versions that were not detected in testing at the previous strength. For example, 2-way testing did not detect 2 out of 9 faulty versions of the *schedule* program. So 3-way testing was performed on these 2 versions, which did not detect any of the two versions. At this point, we stopped testing and started to inspect the testing results.

A. *Replace*

We explained the modeling details of the *replace* program in the previous section. We applied 2-way testing to this program, which had a total of 192 tests. We detected all the 32 faulty versions of this program.

B. *Schedule*

Two programs, *schedule* and *schedule2*, take the following inputs: (1) three non-negative integers representing the number of processes in three different priority queues, *low*, *medium* and *high*; and (2) a list of commands that must be done on queues. The output of these two programs is a list of numbers indicating the order in which the processes exit (from the scheduling system).

For example, consider the first three input parameters which are 3, 2 and 1. Three processes are placed in low priority queue, two processes in medium priority queue, and one process is high priority queue. The id is assigned to the processes by their priority so the 0 is in the high priority queue, 1 and 2 are in medium priority queue and 3, 4 and 5 are in low priority queue.

There are seven commands (1) *new job*: this command has one attribute, *queue*, and adds a new process at the specified priority queue. (2) *upgrade_prio*: it has two attributes, *queue* and *ratio*. This command promotes a process form the specified priority queue to the next higher priority queue. The *ratio* attribute is used to determine which process to be promoted. (3) *block*: this command adds the current process to the blocked queue. (4) *unblock*: this command unblocks a process from the blocked queue. It has one attribute, *ratio*, which is used to determine which process must be unblocked. (5) *quantum_expire*: this command puts the current process at the end of its priority queue. (6) *finish*: this command exits the current process and prints its number. (7) *flush*: this command causes all processes from the priority queues to exit in their priority order.

Two commands, *upgrade_prio* and *unblock*, operate on the *n*-th process where $\mathrm{n} = (\mathrm{int})\,(\mathrm{r} + 1)$ and $\mathrm{r} = (\text{length of queue} * \text{ratio})$.

In our previous example, if a *flush* command (7) is executed, the output is 0 1 2 3 4 5. But, assume that before the *flush* command, a *new job* command (1 3) is executed, where 1 indicates the new job command and 3 indicates the high priority queue. This *new job* command adds a process to the high priority queue. The next available ID, which is 6, is assigned to the new process and the process is placed at the end of the high priority queue, i.e. after process 0. Now, if we execute the flush command, the output will be 0 6 1 2 3 4 5.

TABLE IV THE ABSTRACT MODEL OF SCHEDULE

Parameters	Values
new_process	[0, 1, >1]
new_proc_queue	[low, mid, high]
upgrade_prio	[0, 1, >1]
upgrade_queue	[low, mid]
upgrade_ratio	[0, 1, >1, {r}=0.1, {r}=0.4, {r}=0.5, {r}=0.6, {r}=0.9]
block	[0, 1, >1]
unblock	[0, 1, >1]
unblock_ratio	[0, 1, >1, {r}=0.1, {r}=0.4, {r}=0.5, {r}=0.6, {r}=0.9]
quantum_expire	[0, 1, >1]
finish	[0, 1, >1]
flush	[0, 1, >1]

Table IV shows the input model of the two *schedule* programs. Commands and their attributes are modeled as parameters. Each command parameter has three values, 0, 1 and >1, where 0 means that this command does not appear, 1 means that this command appears once, and >1 means that this command appears more than once. The *priority* attribute of the *new job* command could be one of the three possible queues. But the attribute of *upgrade_prio* could be either low or mid. (Processes in the high priority queue cannot upgrade.)

Two commands *unblock* and *upgrade_prio* are affected by the length of the queues, they select a process based on queue's length and ratio. For these commands, first, we test if the ratio equals to 0, 1, or >1. Then we check that if the number after floating point in $r = (\text{length of queue} * \text{ratio})$ is 1, 4, 5, 6 or 9. These numbers are selected to cover upper limit (9), lower limit (1) and middle of the range (5), and also two numbers (4 and 6) around the middle.

A C++ program was written to create the file that contains commands based on abstract tests. For the initial length of the queues, we randomly selected 60. We fixed >1 values to 2, i.e. if the value of a command is >1, the command appears twice in the file.

Performing 2-way testing detected 7 out of 9 versions of the *schedule* and 3 out of 10 versions of the *shedule2*. In total, 9 versions were not detected. Performing 3-way testing did not detect any more versions. We investigated all versions that were not detected, 8 out of 9 (version 9 of the *schedule* and 7 versions, 1, 4, 5, 6, 8, 9 and 10, of the *schedule2*) can be detected by invalid testing, which as mentioned is not the focus of our study.

For example, version 10 of the *schedule2* was detected by a test case which contains *new_process* or *upgrade_prio* commands with invalid value for the queue attribute (*new_proc_queue* or *upgrade_queue* parameter).

Version 8 of the *schedule* is the only version that was not detected and could not be detected by invalid testing. This version could be detected only when two *upgrade* commands, one *block* command, and one *unblock* command are executed consecutively on one process.

The following example will reveal the bug:

./schedule 2 2 0 <file.txt

There are 4 processes, 0 to 3, two of which, 0 and 1, are in the *mid* priority queue, and the other two, 2 and 3, are in the low priority queue. The high priority queue is empty. Figure 1Figure 2 shows the file that contains 5 commands. The comments explain the state of the system after each command is executed.

In the *schedule* program, each process keeps the id of the queue to which it belongs. The faulty code in the version 8 does not change the queue id of the process after the *upgrade* command (lines #1 and #2). Thus when the process is unblocked (line #4), it is assigned to the wrong queue.

We did not detect this version, because our approach, at this point, does not generate test sequences. Combinatorial test sequence generation is a subject that we plan to study in the future.

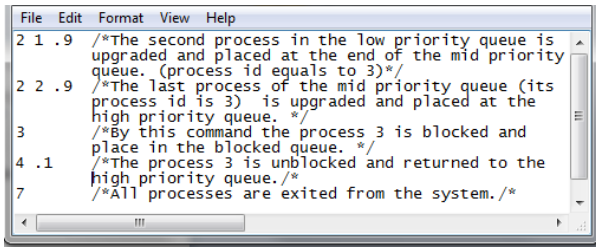

Figure 2 File Example to Detect v8 of schedule

C. *tcas*

This program was previously modeled by Kuhn et al. in [8] [9], based on the specification in [11]. The *tcas* program is an aircraft collision avoidance system, and it takes 12 numbers as input and generates as output one number, which can be 0, 1 and 2.

Table V shows the input model of the tcas program. Some input parameters, e.g., *high_confidence*, *two_of_three_reports_valid*, and c*limb_inhibit*, are boolean values, 0 and 1. Some input parameters, like *alt_layer_value*, are of enum type and have a set of specific values. For the other parameters, the values are identified by analyzing the code and by equivalence partitioning. Note that the input space of this program is not complex, and thus an abstract model is not needed.

According to [8] all 41 faulty versions of *tcas* are detected by the model. The maximum strength to detect all versions is six; we also got the same results.

As discussed in Section F, all faulty versions of the tcas program were detected by 6-way testing. However, the degree of fault is actually more than 6 in all faulty versions. Thus, these faulty versions were actually detected by higher strength combinations that happen to appear in a 6-way testing.

D. *Totinfo*

This program takes as input a file containing one or more tables. The program uses the notions of chi-square and degree of freedom to calculate whether the distribution of the numbers in these tables is logarithm-

TABLE V THE ABSTRACT MODEL OF TCAS

Parameters	Values
cur_vertical_sep	[299,300, 601]
high_confidence	[0, 1]
two_of_three_reports_valid	[0, 1]
own_tracked_alt	[1, 2]
own_tracked_alt_rate	[600, 601]
other_tracked_alt	[1, 2]
alt_layer_value	[0,1, 2, 3]
up_separation	[0, 399, 400, 499, 500, 639, 640, 739, 740, 840]
down_separation	[0, 399, 400, 499, 500, 639, 640, 739, 740, 840]
other_rac	[0, 1, 2]
other_capability	[1, 2]
climb_inhibit	[0, 1]

gamma distribution. The output is the total degree of freedom of rows and columns and chi-square.

We focused on the correctness of the syntax of input parameters instead of the mathematical aspect of the program. The reason is that the logic of the program is very complex and is difficult to understand due to a lack of specification.

We identified a total of 6 parameters related to the syntax input of the program. Parameter *# of tables* can be 0, 1 or more than one. The maximum number of members in a table is 1000. We set the maximum number of rows and columns to 500 and the minimum number of rows and columns to 1. Thus, parameters *# of rows* and *# of columns* have three values, 1, between 2 and 499, and 500.

Parameter *tbl_attr* is identified to define general attributes for tables' elements. One important attribute for the table elements is sign, they can be *positive*, *negative*, *zero*, or *mix*. The number of elements is another attribute we identified for *tbl_attr*. The number of the elements in a table defined by $\#of\ rows \times \#\ of\ columns$; we added *sufficient*, *more than* and *less than enough* values to check that whether the number of elements in the table is consistent with $\#of\ rows \times \#\ of\ columns$.

The *option* parameter models the position in which a comment appears. The *maxline* parameter defines the maximum number of lines in the input file.

A program was written to generate the input tables from the abstract tests. 2-way testing detected 5 out of 23 versions. 3-way testing detected 7 more versions, but 4-way testing did not detect any new version. So, totally 12 out of 23 versions were detected. We investigated the 11 versions which were not detected by the model. All of these versions have faults related to the mathematical aspects of the program, which is out of our testing scope.

TABLE VI THE ABSTRACT MODEL OF TOTINFO

Parameters	Values
#of tables	[0, 1, >1]
#of rows	[1, between 2 and 499, 500]
#of columns	[1, between 2 and 499, 500]
tbl_attr	[sufficient number positive[1], sufficient number negative[2], sufficient number mix[3], sufficient number equal 0[4], more than enough[5], less than enough[6]]
options	[normal, row & column in 2 lines, comment at the beginning, comment in the middle, comment at the end]
maxline	[1, Between 2and 254, 255, 256, 257]

[1]There are $\#of\ rows \times \#\ of\ columns$ positive numbers in the input file.

[2]There are $\#of\ rows \times \#\ of\ columns$ negative numbers in the input file.

[3]There are $\#of\ rows \times \#\ of\ columns$ positive and negative number in the input file.

[4]There are $\#of\ rows \times \#\ of\ columns$ zero in the input file.

[5]There are less than $\#of\ rows \times \#\ of\ columns$ numbers in the input file.

[6]There are more than $\#of\ rows \times \#\ of\ columns$ numbers in the input file.

E. *Printtokens Model*

The goal of the two programs, *printtokens* and *printtokens2*, is tokenizing the input file and determining the type of each token. Token could have one of these types: *identifier*, *special*, *keyword*, *number*, *comment*, *character constant* or *string constant*.

Keyword type includes *and*, *or*, *if*, *xor*, and *lambda*. Special type includes *lparen*, *rparen*, *lsquare*, *rsquare*, *quote*, *bquote*, *comma* and *equalgreater*. Comment is started with semicolon and ended when a new line character is seen. String constant is confined in two double quotations. Character is a token started with #.

To model the system, we divided it into seven subsystems: keyword, special, identifier, number, comment, character and string. By this classification each token type was tested independently from the others. We assumed that the program analyzes each token independent from previous and next token, i.e. the type of the previous or next token does not affect on the analyzing the current token.

Each subsystem has 3 parameters, value, position and number of lines. Keyword model is shown in Table VII, as an example. The *kyw_value* parameter covers all possible values for keyword (corresponding token type in general). An important property for each token type is position, depends on different position of token type the program may behave differently. So for each token type the position property with three values, *begin, middle* and *end*, is added to the model. The last parameter, *# of lines*, checks the behavior of the system when the input file has a single line or multiple lines.

The possible values for some token types, such as keyword and special are explicitly defined in the program specification. But for the others such as identifier, the features and characteristic of its values are described in the specification. For each token type, identifier, number, comment, character and string, we designed an abstract model to define their values. Then after the possible values were defined in the next level they have the same model as keyword. We explain the model of values for three subsystems identifier, number and comment in more details.

TABLE VII THE ABSTRACT MODEL OF KEYWORD

Parameters	Values
kyw_value	[and, or, xor, if, lambda]
position	[begin, middle, end]
# of lines	[1, >1]

TABLE VIII THE ABSTRACT MODEL OF IDENTIFIER VALUES

Parameters	Values
lowercase	[off, on]
uppercase	[off, on]
number	[off, on]
keyword	[off, on]
whitespace	[Space, tab]

TABLE IX THE ABSTRACT MODEL OF NUMBER VALUES

Parameters	Values
#of digits	[1, >1]
begins with zero	[off, on]

TABLE X THE ABSTRACT MODEL OF COMMENT VALUES

Parameters	Values
identifier	[off, on]
keyword	[off, on]
character	[off, on]
string	[off, on]
special	[off, on]
number	[off, on]
comment	[off, on]
whitespace	[Space, tab]

Identifier has different feature such as having uppercase, lowercase, keyword or numbers, a model is designed to cover all features of identifier values (Table VIII) .These features are parameters with two values *off* and *on*, to show weather an identifier contains the parameter or not. The *whitespace* parameter determines whether an identifier separate from next token by *space* or *tab*. Note that we add a constraint to prevent having null identifier. For 2-way test generation, we generate 2-way test set for identifier values model first. The number of tests is 7. Then, we put these seven tests as values in the *value* parameter of the identifier model, and generate 2-way test set for identifier.

For the number model, the characteristics of the number are the number of digit and having zero at the beginning of it. So its model has 2 parameters, Table IX. Note that sign and decimal point do not support by the *printtokes* programs.

The comment model is shown

Table X. We check the behavior of the system when each token type appears as a comment. Also, *whitespace* parameter determines if a comment separate from next token by *space* or *tab*, what would be the behavior of the system. The models of sting and character values are the same as comment.

The 2-way testing detected 2 out of seven versions of the *printtokens* and nine versions out of 10 versions of the *printtokens2*. Note that 2-way test set has only 141 tests.

The programs were tested by 3-way testing, but no new version was detected. So we stopped testing and investigated versions which were not detected. Five versions out of six can be detected by invalid testing. For example, in versions 6 of the *printtokens*, the failure happen when the number of tokens in the input file exceeds the defined value. The second version of the *printtokens* is not detected by invalid testing. The fault in this version is adding code. The adding code is reached when there is a **i** token in the input file.

F. *Discussion*

After testing programs using the combinatorial technique, we investigated the faults detected by our model to ensure that the fault is caused by the interaction between input parameters. In order to do that we introduce the notion of degree of fault or fault strength which is defined to be the minimum number of parameters that must be involved to trigger the fault.

As a t-way test set contains all t-way combinations, it is guaranteed to detect a faulty version if the strength of the fault does not exceed t. But it is also possible that a t-way test set detects a version whose degree of fault is higher than t. This is because the test set may contain the inducing combination (in which more than t parameters are involved) by chance.

In Table XI, we classified the degree of fault for all detected versions. For example, in the *schedule* program, the model detected a total of 7 versions. The fault strength in five of these versions is 2. In the two remaining versions, one of them is 3 and another one is 4.

To define the degree of fault, we used the concept of inducing combination. An inducing combination is a combination of parameter values such that all test cases containing this combination fail. The length of the minimum inducing combination shows the degree of fault.

We used a tool called BEN [3] to find minimum inducing combinations. BEN takes a t-way test set as input and generates a ranking of t-way combinations based on their likelihood to be inducing combinations. BEN has been shown very effective in identifying inducing combinations [3]. However, BEN is heuristic by nature and thus does not guarantee to always find minimum inducing combinations. This should be taken into account when reading the results in Table XI. We are not aware of any method that can precisely determine the degree of a fault.

For example seven versions of *schedule* are detected by 2-way test sets. BEN finds an inducing combination for five of them, so the degree of fault is 2 for these versions. For the two other versions BEN did not find an inducing combination, we used a 3-way test set. BEN finds an inducing combination for one of them. We then used a 4-way test set for the last version, which found an inducing combination.

Since there is a probability that the fault is not due to any parameter interaction, we need to check whether only one parameter is involved in the fault. BEN has a feature to derive inducing combinations with smaller size than t. We used this feature on 2-way test sets, to derive one-way inducing combination. In ten versions of the *replace* the

TABLE XI FAULT CLASSIFICATION OF DETECTED VERSIONS

Program	#faulty versions with degree of fault							
	1	*2*	*3*	*4*	*5*	*6*	*Beyond 6*	*sum*
print_tokens	0	0	2	0	0	0	0	2
print_tokens2	0	6	3	0	0	0	0	9
replace	10	7	2	0	0	0	13	32
schedule	0	5	1	1	0	0	0	7
schedule2	0	3	0	0	0	0	0	3
tcas	0	0	0	0	0	0	41	41
totinfo	0	0	2	1	6	3	0	12

TABLE XII FAULT CLASSIFICATION BASED ON TEST STRENGTH

program	Test strength	# of detected versions with the same or lower strength	# of detected versions with higher strength	total	Total not detected
print_tokens	2	0	2	2	5
print_tokens2	2	6	3	9	1
replace	2	17	15	32	0
schedule	2	5	2	7	2
schedule2	2	3	0	3	7
tcas	2	0	9	41	0
	3	0	13		
	4	0	14		
	5	0	4		
	6	0	1		
totinfo	2	1	4	12	11
	3	1	6		

- If a t-way test detects a version, the version does not show in the result of (t+1)-way test.
- All 1-way and 2-way faulty versions of replace are detected in 2-way test set.

degree of fault was 1. Table XI shows that most faults are interaction faults.

In 13 versions of *replace* and 41 versions of *tcas*, BEN cannot identify inducing combinations in the 6-way test sets, so the degree of fault is more than 6 for these versions. Note that in the *replace* all 13 versions and in the *tcas* 9 of these versions are detected by 2-way testing. A 2-way test set is not guaranteed to detect these versions, since it is not guaranteed to cover all combinations for t > 2, and the versions are detected accidentally.

We show the strength of fault for detected versions in respect to the test strength in Table XII. The second column shows the test strength at which the faulty versions were detected. The third one shows the number of faulty versions that were detected by the test set, and the combinatorial test set guarantees to detect them, since their fault strength is equal or less than the test strength. The forth column shows the number of detected versions with higher fault strength than test strength, which are detected by chance.

For example, by applying 2-way testing to all faulty versions of the *replace* program, we detected not only 17 versions whose degree of fault is 1 or 2, but also 13 versions whose degree of fault is higher than 6.

Another point to note is that, in each step we excluded detected versions in the next step. For example, in the *totinfo* program 5 versions were detected by 2-way testing. One of these 5 versions has the same degree of fault as the test strength, i.e., 2, and the other four versions have the degree of fault higher than the test strength. For the next step we excluded all five versions from testing and we applied 3-way testing only on versions which were not detected.

G. *Comparison*

In this section, we show the effectiveness of combinatorial testing by comparing it with random testing. We generated a random test suite corresponds to each combinatorial test set which was used in the previous section. The random test suite and its corresponding combinatorial test set have the same number of tests. For example, the 2-way combinatorial test set for *printtokens* program has 141 tests; thus 141 tests are generated for random testing.

For random test generation, we used the models which were described. Since the subject programs have complex input spaces, we cannot apply random testing without any abstraction. For instance, the first input parameter in the *replace* program is a regular expression; generating valid random regular expressions is impractical.

Our random test generation approach is as follows. For programs whose models do not have any constraint, *schedule*, *schedule2*, *tcas* and *totinfo*, a random value is selected for each parameter in a test. For *printtokens,* we generate the same number of tests as a 2-way test set for each subsystem. If the value parameter comes from the model, such as *identifier*, first we randomly generate a test for value, and then for the subsystem.

If a model has constraints, random selected values may create invalid tests. We avoided invalid tests using the following algorithm. In the *replace* program, constraints are related to the position of elements. There are 4 parameters related to substitution. At most one of them can be *begin* and also at most one can be *end*. Note that it is possible for a test case to not include *begin* or *end* .

To generate random values for substitution related parameters (*sub_character*, *sub_@n*, *sub_@character*, *sub_&*), we define which parameter should appear at the beginning and which one at the end, randomly. A number between 0 and 4 (number of parameters, *sub_character, sub_@n, sub_@character* and *sub_&,* plus 1) are selected randomly. This number is used to select the parameter whose value should be *begin* and appearing at the beginning. If 0 is selected, the first parameter, *sub_character* is set to *begin*, and so on. If 4 is selected, none of the parameters would have *begin* value. Similarly, we select the parameter that should appear at the end. For other parameters, *off* or *middle* is selected randomly. The same approach is used for parameters which are involved in the pattern.

Table XIII compares the results of combinatorial and random testing. The second column shows the number of tests in the test sets, third and forth columns are shown the strength and the number of detected versions in combinatorial test set. The last column shows the number of detected versions in random test sets. According to the table, the result of random testing is different in different programs. In the two *schedule* programs, *schedule* and *schedule2*, combinatorial testing and random testing have the same results, 7 versions in the *schedule* and 3 versions in the *schedule2* were detected.

TABLE XIII COMPARE RANDOM TESTIN AND COMBINATORIAL TESTING

Program	#tests	Combinatorial		Random
		Strength	*#detected version*	*#detected version*
print_tokens	141	2-way	2	1
print_tokens2	141	2-way	9	9
replace	192	2-way	32	17
schedule	64	2-way	7	7
schedule2	64	2-way	3	3
tcas	100	2-way	9	7
	400	3-way	13	14
	1363	4-way	14	6
	4222	5-way	4	12
	10843	6-way	1	2
totinfo	30	2-way	5	2
	156	3-way	7	5

But in the *replace* program, random testing detected 17 versions compared to 32 versions in combinatorial testing.

In the *tcas* program, combinatorial test set and random test set detected all 41 faulty versions. But combinatorial test can detect more versions by using fewer tests. Combinatorial test sets, 2-way, 3-way and 4-way, detected 36 versions, but random test set with the same number of tests detected 27 versions.

IV. RELATED WORK

First, we review existing work on input parameter modeling for combinatorial testing. Grindal and Offutt [4] presented a structured method for input parameter modeling. Their method provides guidelines for defining parameters, values, constraints and relations. We followed this method, whereas applicable, in our experiments.

Several common patterns were reported for combinatorial models [14][15]. These patterns include optional values, multi-selection, ranges and boundaries, order and padding, redundant interactions, and auxiliary aggregates or commonality. We used similar ideas for optional values, order and padding, and multiplicity patterns in our experiments. For example, the optional values pattern occurred in the *replace* program. We added the *off* value for each optional parameter.

Segall et al. suggested two constructs, called counters and properties, to model high-level constraints [17]. Some abstract parameters, e.g., the position parameter, identified in our experiments can be considered as properties of a concrete parameter. However, these parameters are not used to facilitate constraint specification in our experiments.

Second, we review existing work on empirical studies on combinatorial testing. We focus on these controlled studies. Dalal et al. [2] reported four relatively large applications that are modeled for combinatorial testing. They reported the number of failed tests and the number of different types of failures that were detected. They showed that combinatorial testing was more effective than traditional testing methods. The difference between their approach and our work is that they did not identify abstract parameters and values. In addition, their subject programs contain real faults, instead of seeded faults. [2]

Kuhn et al. studied several fault databases and found that all the faults in these databases are caused by interaction of no more than six parameters [8][9]. This study did not perform actual combinatorial testing on the subject systems.

Schroeder et al. compared combinatorial testing to random testing in a controlled study [13]. They selected two software applications used in their laboratory and used faults that are manually seeded by a graduate student. In contrast, the Siemens suite used in our experiments is a third-party benchmark that has been used to evaluate many testing techniques [17]. We also used faults that come with the Siemens suite.

In [7], Kuhn et al. applied combinatorial testing to a multicomputer network simulator. They compared combinatorial testing to random testing in terms of the number of deadlocks that can be detected by both approaches. The modeling process was not explained in [7].

In [11][12], combinatorial testing was compared to several prioritization techniques and random testing. The experiments were done on two programs *flex* and *make* from SIR [16] repository. The results showed there was no significant difference between combinatorial testing and random testing. The details about the programs models were, however, not, reported in the paper.

V. CONCLUSION

In this paper, we presented a three-step approach to apply combinatorial testing. First we create an abstract model for the system. Then, based on this model, a combinatorial abstract test set is generated. The last step derives a set of concrete tests from these abstract tests. We reported our experiments in which we modeled the seven programs in the Siemens suite and applied combinatorial testing to these programs. The details of the abstract model and the results of applying combinatorial testing are presented in the paper. The results show that combinatorial testing can detect most faulty versions of the Siemens programs, and is more effective than random testing.

To better understand the effectiveness of combinatorial testing, we distinguished faults guaranteed to be detected by t-way testing from faults detected incidentally. A fault is detected incidentally by a t-way test set if the degree t' of the fault is higher than t, but the t-way test set happens to contain a t'-way combination that can trigger this fault. In our experiments, we observed that t-way testing often detected some faults incidentally, i.e., the degrees of these faults were higher than t. In particular, for the *tcas* program, all the faults were detected incidentally. This suggests that a t-way test set can be potentially more effective if it covers more higher-strength combinations, in addition to all the t-way combinations.

In the future, we plan to conduct more empirical studies on larger and more complex programs. We believe this research will provide guidance for practitioners to apply combinatorial testing in practice.

ACKNOWLEDGMENT

This work is supported by two grants (70NANB9H9178 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST).

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

REFERENCES

[1] Advanced Combinatorial Testing System (ACTS), 2010. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.

[1] M. N. Borazjany, Y. Linbin, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial Testing of ACTS: A Case Study. In Proc. of the 5th IEEE International Conference on Software Testing, Verifcation and Validation, ICST, pages 591-600, Montreal, Canada, 2012.

[2] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In Proceedings of the 21st international conference on Software engineering , pages 285-294, New York, USA, 1999.

[3] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. 2012. Identifying Failure-Inducing Combinations in a Combinatorial Test Set. In Proceedings of International Conference on Software Testing, Verification and Validation, IEEE Computer Society, Washington, DC, USA, pages 370-379, 2012.

[4] M. Grindal , J. Offutt, Input parameter modeling for combination strategies, Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, pages 255-260, Innsbruck, Austria,2007.

[5] M. Grindal, J. Offutt, and S. F. Andler. 2005. Combination Testing Strategies: A Survey. Journal of Software Testing, Verification and Reliability vol. 15, no. 3, pp. 167-199, 2005.

[6] R. Krishnan, S. Murali Krishna, and P. Siva Nandhan. Combinatorial testing: learnings from our experience. ACM SIGSOFT Software Engineering Notes, v.32 n.3, May 2007.

[7] D. R. Kuhn, R. Kacker, Y. Lei. Combinatorial and Random Testing Effectiveness for a Grid Computer Simulator. presented at the Mod Sim World, Virginia, USA, 2009.

[8] D. R. Kuhn and V. Okum. 2006. Pseudo-Exhaustive Testing for Software. 30th NASA/IEEE Software Engineering Workshop, pages 153-158, April 2006.

[9] D. R. Kuhn, D. Wallace, and A. Gallo, Software Fault Interactions and Implications for Software Testing, IEEE Transactions on Software Engineering, 30(6): 418-421, 2004.

[10] C. Nie and H. Leung. 2011. A survey of combinatorial testing. ACM Computing Surveys (CSUR), v.43 n.2, pages 1-29, January 2011

[11] V. Okun, Specification Mutation for Test Generation and Analysis, PhD Dissertation, University of Maryland, 2004

[12] X. Qu, M. Cohen, and K. Woolf, Combinatorial interaction regression testing: A study of test case generation and prioritization. In Proceedings of the IEEE International Conferance on Software Maintenance (ICSM). IEEE Computer Society, 413–418, 2007.

[13] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. Proceedings of International Symposium on Empirical Software Engineering, pages 49-59, August 19-20, 2004.

[14] I. Segall, R.Tzoref-Brill, and A. Zlotnick. 2012. Common Patterns in Combinatorial Models. In Proc. of the 5th IEEE International Conference on Software Testing, Verifcation and Validation, ICST, pages 624-629, Montreal, Canada, 2012.

[15] I. Segall, R. Tzoref-Brill, and A. Zlotnick. Simplified Modeling of Combinatorial Test Spaces. In Proc. of the 5th IEEE International Conference on Software Testing, Verifcation and Validation, ICST, pages 573-579, Montreal, Canada, 2012.

[16] Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php, 2012.

[17] E. Wong and V. Debroy, A survey on software fault localization, Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, Nov. 2009.