2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Combinatorial Testing of ACTS: A Case Study

Mehra N.Borazjany, Linbin Yu, Yu Lei
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019, USA
{mehranourozborazjany,linbinyu,ylei}@uta.edu

Raghu Kacker, Rick Kuhn
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, Maryland 20899, USA
{raghu.kacker,kuhn}@nist.go

Abstract— **In this paper we present a case study of applying combinatorial testing to test a combinatorial test generation tool called ACTS. The purpose of this study is two-fold. First, we want to gain experience and insights about how to apply combinatorial testing in practice. Second, we want to evaluate the effectiveness of combinatorial testing applied to a real-life system. ACTS has 24637 lines of uncommented code, and provides a command line interface and a fairly sophisticated graphic user interface. The main challenge of this study was to model the input space in terms of a set of parameters and values. Once the model was designed, we generated test cases using ACTS, which were then later used to test ACTS. The results of this study show that input space modeling can be a significant undertaking, and needs to be carefully managed. The results also show that combinatorial testing is effective in terms of achieving high code coverage and fault detection.**

***Keywords*—Combinatorial Testing; Input Parameter Modeling; Software Testing.**

I. INTRODUCTION

Software failures are often the result of a faulty interaction between input parameters. Empirical studies show that most faults are caused by interactions among six or fewer parameters [16]. Combinatorial testing, which has proven very effective in fault detection, is a testing strategy that applies the theory of combinatorial design to test software systems. Given a system under test with *k* parameters, t-way combinatorial testing requires all combinations of values of t (out of k) parameters be covered at least once, where *t* is usually a small integer. If test parameters are modeled properly, all faults caused by interactions involving no more than *t* parameters will be detected. Combinatorial testing can significantly reduce the cost of software testing while increasing its effectiveness.

Input Parameter Modeling is an important step in combinatorial testing. An input parameter model (IPM) contains a set of parameters, each of which has a set of possible values [7]. There are important design decisions and tradeoffs to be made in the modeling process. Different testers may come up with different models, depending on creative choices and experience [1]. Grochtmann and Grimm [6] mentioned that finding parameters and values is a creative process that can never be fully automated. Several methods could be used for IPM, such as Category Partition [7] or Classification Trees [6]. A basic eight-step process that is custom-designed to be used with combinatorial testing is suggested in [1].

In this paper we present a case study in which we applied combinatorial testing to a combinatorial test generation tool called ACTS [21]. ACTS is developed jointly by the US National Institute Standards and Technology and the University of Texas at Arlington, and currently has more than 900 individual and corporate users. This study was conceived when a user of ACTS asked the question: Have you tested ACTS using ACTS? The objective of this study is two-fold. First, we want to gain experience and insights about how to apply combinatorial testing in practice. Second, we want to evaluate the effectiveness of combinatorial testing applied to a real-life system. Compared to extensive work that has been reported on the theoretical side, there is a lack of empirical studies and experience reports on applying combinatorial testing to real-life systems [17].

The results of our study indicate that combinatorial testing is very effective. In our study, we generated a total number of 1105 tests, and the execution of these tests achieved about 88% statement coverage, and detected 15 bugs in a rather mature system.

The remainder of this paper is organized as follow. Section 2 briefly reviews the existing work on input parameter modeling and combinatorial testing. Section 3 gives a high-level introduction to the ACTS tool. Section 4 describes our approach to modeling input parameters for ACTS. Section 5 reports the experimental results. Section 6 provides concluding remarks and our plan for future work.

II. RELATED WORK

We discuss related work in two areas, including input parameter modeling (IPM) and empirical studies on combinatorial testing.

Several approaches, e.g., Category Partitioning [7] and Classification Tree [6], have been reported for the general problem of identifying parameters and parameter values. These approaches can be applied to combinatorial testing. Grindal and Offutt suggest an input parameter modeling method that is specifically designed for combinatorial testing [1]. This method provides more guidance in the parameter and parameter value selection. Beizer [10], Malaiya [11], and Chen et al. [5] also addressed the problem of parameter and value selection but did not describe the complete processes.

Several empirical studies of combinatorial testing have been reported that applied combinatorial testing on major features of a mobile phone application testing [18], an email system testing [19], satellite communications system testing [22], configuration testing [23], browser compatibility testing [24], network interface testing [25], and protocol testing [26]. The studies show that combinatorial testing is very effective and can be applied to a wide variety of applications. The purpose of these studies was to evaluate the effectiveness of combinatorial testing, whereas we also try to gain experience and insights about the input modeling process.

III. ACTS

ACTS is a test generation tool for constructing *t*-way combinatorial test sets. Currently, it supports *t*-way test set generation with *t* up to 6. The tool is implemented in Java and provides both command line and graphical user interfaces. In the following, we briefly discuss the core features in ACTS.

- *T-Way Test Set Generation*: A system configuration is specified by a set of parameters and their values. A test set is a *t*-way test set if it satisfies the following property: Given any *t* parameters, every combination of values of these *t* parameters is covered in at least one test in the test set. Several test generation algorithms are implemented in ACTS. These algorithms include IPOG, IPOG-D, IPOG-F, IPOG-F2, and PaintBall. ACTS supports two test generation modes, namely, scratch and extend. The former allows a test set to be built from scratch, whereas the latter allows a test set to be built by extending an existing test set.

- *Mixed Strength* (*or Relation Support*): Relations are groups of parameters with different strengths. ACTS allows arbitrary parameter relations to be created, where different relations may overlap or subsume each other. In the latter case, relations that are subsumed by other relations will be ignored by the test generation engine.

- *Constraint Support*: Some combinations are not valid and must be excluded from the resulting test set. ACTS allows the user to define invalid combinations by specifying constraints. The specified constraints are taken into account during test generation so that the resulting test set will cover combinations that satisfy these constraints.

- *Coverage Verification*: This feature is used to verify whether a test set satisfies *t*-way coverage, i.e. whether it covers all the *t*-way combinations.

IV. Input Parameter Modeling

Testing methods are generally categorized as either white-box or black-box testing. In white-box testing, expected results are identified from the specification but inputs are derived from the implementation. In black-box testing, both input and expected results are identified from the specified functional requirements. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing, a testing method emphasized on executing the functions and examination of their input and output data.

The first step is to select the testing method. The functionality testing is used in this experiment, as it also suggested by [1,12,13].

The second step was to identify the test parameters based on system characteristics. We added "M_" at the beginning of all identified parameters to make a distinction between our model parameters versus those in the ACTS tool system configurations. We call our model parameters the Test Factors.

The next step was to identify the test values. Valid-values boundary-values and invalid-values are typically suggested to identify the values for the factors. In this experiment we identified both valid and invalid values. We used valid values to perform the normal functionality testing and invalid values to perform robustness testing. We called our model values the Test Values.

Next we discovered the relationship between the identified test factors. Then we derived the abstract model and finally introduced concrete values to the model and generated test cases to perform both functionality testing and robustness testing.

One of our design decisions in this experiment was the strength of the test cases. We started from 2-way testing and then we extended the generated test cases to perform 3-way testing. This helped us to evaluate the impact of 2-way testing and 3-way testing on code coverage and fault detection.

Another design decision involves introducing constraints to the model to support robustness testing. The ACTS tool does not support robustness testing; therefore, we manually introduced some constraints to ensure that in each test we have only one invalid value among each combination of values.

We will present our modeling process in the following paragraphs. First we will discuss our model for system under test. This is one of our important models as it contains system configurations and core features of the ACTS tool. Second, our command line interface model will be presented and finally our graphical user interface model will be discussed.

A. System Under Test Modeling

System under Test (SUT) contains the configuration information of the system e.g. Parameters, Relations, and Constraints. In order to model the SUT, we have to model its components. The models for M_Parameters, M_Relations, and M_Constraints are as follows:

- ***M_Parameters***: The M_Parameters is defined to model the parameter component in the ACTS. The parameter itself has three parts; name, value, and type. Currently, four types of parameters are supported: Enum, Boolean, Range, and Integer. The Range type is basically a subset of Integer type. Entering a range is a feature in a GUI for facilitating entering values that are in range. It does not affect the system since it interprets to integer and then stores. However when we test the normal functionality of the GUI we consider the Range type as well.

First, for each individual parameter, we identified two factors; value per parameter and type. The name factor is not important from the functionality perspective; therefore, we did not consider it in our model. The M_Parameters factors are shown in Table 1.

Table 1: M_Parameters test factors for one parameter

Type	Value per parameter
Boolean	Invalid
Integer	[true,false] (default)
Range	One or more (valid values)
Enum	

Next, we discovered the relations between these factors. There are some constraints between Type and value of a parameter, e.g. the only valid value for a boolean type parameter is the default value which is [true,false]. If the type is Enum, its value is either an invalid value such as a space character in robustness testing or a valid value in functionality testing. We want to ensure that for each parameter we cover all its type-value combinations at least once. All possible type-value combinations of the M_Parameters are shown in Table 2.

Table 2: Abstract IPM: type-value combinations of M_Parameters

Type-Value combinations
Boolean type with Invalid value
Boolean type with Default value
Boolean type with one or more value
Integer type with Invalid value
Integer type with one or more value
Enum type with Invalid value
Enum type with one or more value

Some of these combinations are useful for functionality testing and others for robustness testing. The robustness testing for the command line interface and the graphical user interface (GUI) are different in some cases. E.g. in the GUI when we select a Boolean type parameter, we cannot select any value, since its feature is disabled. The value is [true,false] by default. This is incorrect in the command line; therefore, we applied this combination to perform robustness testing in the command line interface. The gray rows in Table 2 show the combinations that are only applicable for robustness testing of the command line.

Also this model is an abstract model and we need concrete values to perform functionality testing. The Integer were selected so that we have positive, zero, and negative values in our system. The value for Boolean type is a system defined value and states as [true, false] by default. The values for Enum parameters were selected so that we have a large and small number of values in our system. Enum types in ACTS will accept any character but space. So we will use the space as an invalid value in robustness testing.

In the following examples we assigned concrete values to our abstract model:

- Integer parameters with valid values:
  - num1:[-1000, 10000]
  - num2:[-2, -1, 0, 1, 2] (Range)
- Boolean parameters with Default values:
  - bool1:[true,false]
- Enum parameters with one or more values:
  - Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
  - Enum2:[1, 2]

Afterward, multiple parameters are taken into account. Based on the ACTS specification, the system under test at should have at least two parameters. We tested the system with valid, invalid, and boundary numbers of parameters. We did not find any relation between the number of parameters and the parameter types; therefore we decided to not perform testing on all the different combinations between them, whereas our goal is to cover all types of parameters at least once in the system under test. We decided to select one Integer type and one Enum type parameter when the number of parameters was two in the test, and selected at least one of each type when the number of parameter was three or more to accomplish our goal. The test factors for multiple parameters are shown in Table 3.

Table 3: M_Parameters test factors for multiple parameters

Number of parameters	Parameter Type
Invalid (0 or 1)	Any type
Two	One Integer and one Enum
Three or more	At least one of each type

Finally, based on the information obtained, we generated executable test cases with concrete values. The following example is a parameter component of a system with seven parameters which contains all the parameter types:

Abstract test case:
Number of parameters: ***Three or more***
Parameter type: ***At least one parameter of each type***
num1:[-1000, 10000]
num2:[-2, -1, 0, 1, 2]
bool1:[true,false]
bool2:[true, false]
Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
Enum2:[1, 2]
Enum3:[#]

- ***M_Relations***: The ACTS tool allows arbitrary relations between parameters to be created, where different relations may overlap or subsume each other or may subsume the default relation.

First we identified test factores for the M_Relations. The ACST has two types of relations; default and user-defined. "Default" is the default relation of the system. This relation is not removable and it contains all of the system parameters and the current strength of the system. Also this relation will be automatically added to the system under test. The type and strength are two test factors for the M_Relations. Strength can be a number from 2 to 6 but we only performed our test on 2, 3, and 6. 2 and 6 are boundary values. The test factors for the M_Relations are shown in Table 4.

Table 4: M_Relations test factors for one relation

Type	Strength
Default	2
User-defined (valid parameters)	3
User-defined (invalid parameters)	6

The robustness testing for the command line interface and graphical user interface (GUI) are not the same in M_Relations. The user in the command line interface allows entering a relation to reference the parameters that do not exist in the system.

At this time, we identified the test factors of multiple relations. Based on the ACTS specification when the user adds the user-defined relations to the system, three different situations may occur. Because the default relation is not removable, the user-defined relations will always overlap with the default relation. They may also overlap with each other: “Overlap”, or subsume each other: “Subsume”, or subsume the default relation: “Subsume-default”. The test factors for the user-defined relations are shown in Table 5.

Table 5: M_Relations test factors for user-defined relations

Number of user-defined relations	Relation between user-defined and default relations
0	Overlap
1	Subsume
Two or more	Subsume the default

Our goal was to cover all of the different relations in the system under test. When the number of user-defined relations is zero it means that the system contains only the “default” relation. When the number of user-defined relations is one this means that the system contains two relations; the default relation and the user-defined relation. In this condition, we introduced a user-defined relation that subsumes the default relation, “subsume-default”. When the number of user-defined relations is two or more, the system contains three or more relations; the default relation and two or more user-defined relations. In this condition, we introduced some user-defined relations that “subsume” or “overlap” each other to accomplish our goal.

An example of different relations in a system with the above mentioned values is shown in Table 6.

Table 6: Examople of M_Relations values (default strength 4)

I relation values	Example
default	[4,(bool1, bool2, Enum1, Enum2, num1, num2)]
Subsume-default	[4,(bool1, bool2, Enum1, Enum2, num1, num2)] (default) [5,(bool1, bool2, Enum1, Enum2, num1, num2)]
Overlap	[2,(bool1, bool2, Enum1)] [2,(Enum1, Enum2, num1)]
Subsume	[3,(bool1, bool2, Enum1, Enum2, num1)] [2,(bool1, bool2, Enum1, Enum2, num1)]

The numbers in the bracket represent the strength while the symbols in are a list of parameter names that interact with each other. The default strength in this example is 4 as shown in the first row. The second row shows a relation that subsumes the default relation in row one. The third and fourth rows show the relations that overlap or subsume each other respectively.

- ***M_Constraints***: The M_Constraints is defined to model the constraint component in the ACTS. Currently, three types of constraints are supported: Boolean, Relational, and Arithmetic. Each type will cover some symbols (operators) shows in Table 7.

Table 7: Operators per constraint type

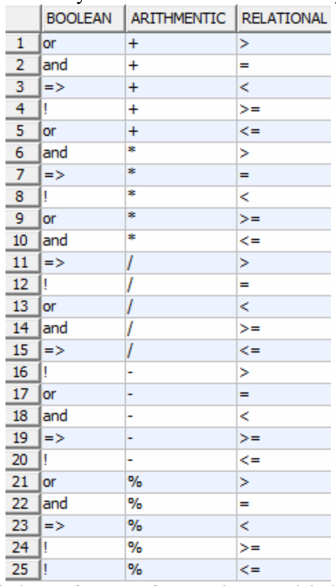
Boolean	Relational	Relational
or	+	=
and	*	>
=>	/	<
!	-	≥
	%	≤

In order to have a meaningful constraint we need to generate a finite combination of symbols (operators) that are well-formed according to applicable rules. We used ACTS to generate all possible 2-way combinations between these three types of operators. ACTS generated 25 different combinations as shown in Table 8. For example three operators in the first row are or, +, and >. We manually generated a constraint that covers all of them e.g. p1+p2>1 or p3; p1 and p2 are two Integer type parameters and p3 is a Boolean type parameter. We generated 25 different constraints to cover all the different 2-way combinations between the different types of constraints.

As this model is an abstract model and we also need concrete values to perform testing. We used valid parameters to generate the constraints in normal functionality testing and one invalid parameter per constraint in robustness testing. An invalid parameter in this case is a parameter that is either not introduced to the system at all, or whose type does not match with its operator type, e.g. a Boolean type parameter and the arithmetic operator.

Afterward multiple constraints were taken into account. The test factors for multiple constraints are shown in Table 9.

Table 8: 2-way combinations of constraints types



	BOOLEAN	ARITHMENTIC	RELATIONAL
1	or	+	>
2	and	+	=
3	=>	+	<
4	!	+	>=
5	or	+	<=
6	and	*	>
7	=>	*	=
8	!	*	<
9	or	*	>=
10	and	*	<=
11	=>	/	>
12	!	/	=
13	or	/	<
14	and	/	>=
15	=>	/	<=
16	!	-	>
17	or	-	=
18	and	-	<
19	=>	-	>=
20	!	-	<=
21	or	%	>
22	and	%	=
23	=>	%	<
24	!	%	>=
25	!	%	<=

We identified three factors for testing multiple constraints. The system under test can have zero, one, or multiple constraints. In addition, adding constraints to the system may introduce unsolvable constraints; therefore, the constraints are not always solvable.

Table 9: M_Constraints test factors for multiple constraints

Number of constraints for each test	constraints relation	Satisfiability
O	related	savable
1	Not_related	unsolvable
Multiple		

Furthermore, it is important to consider the relationship between different constraints. The constraints can be either related or not. The constraints are related if they share at least one parameter. The constraints are not-related if they don’t share any parameter.

The bellow example demonstrates the related constraints (The constraints number 1 and 2 share the parameter n2).

```
1.  (n2 >100) => !b2
2.  e1="1" => !(n2 >100)
```

The bellow example demonstrates the not-related constraints.

```
1.  (n2 >100) =>!b2
2.  e1="1" => !b1
```

These factors are independent and so we don't need to find the different combinations between them. However we need to consider them at least once during our testing process.

Finally based on the information obtained, we generated executable test cases with concrete values. The following example is a system with six parameters and five solvable related constraints in which the constraints cover the rows number 2, 7, 15, 17, and 23 of Table 8:

```
Abstract test case:
Number of parameters: Three or more
Parameter type: At least one parameter of each type
Number of constraint: multiple
Constraint relation: related
Satisfiability: solvable
num1:[-1000, 10000]
num2:[-2, -1, 0, 1, 2]
bool1:[true,false]
bool2:[true, false]
enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
enum2:[1, 2]
enum2="1" && num2+ num1=9999
(num1*num2= 1000) => bool1
num2/num1 <=500 => bool2
enum1="v1"|| num2-num1=9998
 num1%num2<900 => num2<0
```

• ***M_SUT***: As we mentioned before system under test (SUT) contains the configuration information of the system parameters, relations and constraints. In the previous sections we identified test values for each of these components; M_Parameters, M_Relations, and M_Constraints. We combined them to form the M_SUT model. The M_SUT factors and values are shown in Table 10.

Table 10: M_SUT test factors and values

M_SUT	
Test Factors	**Test Values**
M_Parameters	Invalid
	Two *(1 Integer,1 Enum)*
	Three or more *(at least 1 Integer,1 Enum, 1 Boolean)*
M_Relations	Invalid parameter (*just in CMD interface)*
	Default relation
	Two *(default and subsume-default)*
	Multiple relations *(default plus at least 2 subsume)*
	Multiple relations *(default plus at least 2 overlap)*
M_Constraints	None
	Unsolvable
	Invalid
	One
	Multiple not-related constraints
	Multiple related constraints

We decided that there is no interaction between M_SUT factors; therefore, covering each value once would be sufficient. We produced the abstract model of M_SUT which is shown in Table 11. In total, 8 different system configurations have been identified for M_SUT, four of which were used in robustness testing.

Table 11: Abstract IPM of M_SUT

M_ Parameters	M_ Relations	M _Constraints	M_SUT
Two	Two	Multiple not-related	2P_2R_multi-nC
Multiple	Multiple	Multiple related	multiP_multiR_multi-rC
Multiple	Multiple	One	multiP_multiR_oneC
Two	Default	None	2P_2R_noC
Invalid	Default	One	InvalidP
Two	Invalid	One	InvalidR
Two	Default	Invalid	InvalidC
Two	Default	Unsolvable	UnsolvedC

An example of a system under test with six parameters, multiple relations, and multiple related constraints is shown in Table 12(a). An example of SUT with "Invalid constraint" is also shown in Table 12 (b).

Table 12: Example of a SUT

a. M_SUT with multiple parameters, multiple relations and multiple related Constraints (multiP_multiR_multi-rC)

```
    Default degree of interaction coverage: 4
    Number of parameters: 6
Parameters:
    num1:[-1000, -100, 1000, 10000]
    num2:[-2, -1, 0, 1, 2]
    bool1:[true, false]
    bool2:[true, false]
   Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
   Enum2:[1, 2]
Relations :
    [4,(bool1, bool2, Enum1, Enum2, num1, num2)]
    [5,(bool1, bool2, Enum1, Enum2, num1, num2)]
   [2,(bool1, bool2, Enum1)]
    [2,(Enum1, Enum2, num1)]
     [3,(bool1, bool2, Enum1, Enum2, num1)]
Constraints :
    enum2="1" && num2+ num1=9999
    (num1*num2= 1000) => bool1
    num2/num1 <=500 => bool2
    enum1="v1"|| num2-num1=9998
     num1%num2<900 => num2<0
```

b. M_SUT with invalid constraint (num3 doesn't exist)

```
    Default degree of interaction coverage: 4
    Number of parameters: 6
    Number of configurations: 0
Parameters:
    num1:[-1000, -100, 1000, 10000]
    num2:[-2, -1, 0, 1, 2]
    bool1:[true, false]
    bool2:[true, false]
   Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
   Enum2:[1, 2]
Relations :
    [4,(bool1, bool2, Enum1, Enum2, num1, num2)]
    [5,(bool1, bool2, Enum1, Enum2, num1, num2)]
Constraints :
    (num1*num2>num2+100) => bool2!=bool1
    num2/num1 >=10 => !bool2
    num1%num2<=3 => num1<4
    bool1 =>Enum1="v1"
    Enum2="1" && Enum1="v2" => num2=2 || num3=0
```

The factors discussed in the above paragraphs are common between two different interfaces of ACTS. The following paragraphs; however, will identify the specific factors and values for the command line interface and the GUI interface.

B. Command Line Interface Modeling

The various options are available in command line interface as shown in Table 13. There are several test generation algorithms implemented in ACTS. The user has to select one of these algorithms in order to generate the tests. "M_Algorithm" would be chosen as one of our factors with the domain value of [IPOG, IPOG-D, IPOG-F, IPOG-F2, PaintBall]. The IPOG algorithm is the most commonly used algorithm; therefore, in this experiment we performed our test on the IPOG and fixed the value of M_Algorithm to "IPOG". Covering IPM for other algorithms will be one of our future works. Also, ACTS supports two test generation modes, scratch and extend. Obviously "M_mode" is another factor with the domain value of [scratch, extend].

Table 13: Command-Line IPM

Test Factors	Test Values	Description
M_mode	scratch	generate tests from scratch (default)
	extend	extend from an existing test set
M_algo	ipog	use algorithm IPO (default)
M_fastMode	on	enable fast mode
	off	disable fast mode (default)
M_doi	specify the degree of interactions to be covered	
M_output	numeric	output test set in numeric format
	nist	output test set in NIST format (default)
	csv	output test set in Comma-separated values format
	excel	output test set in EXCEL format
M_check	on	verify coverage after test generation
	off	do not verify coverage (default)
M_progress	on	display progress information (default)
	off	do not display progress information
M_debug	on	display debug info
	off	do not display debug info (default)
M_randstar	on	randomize don't care values
	off	do not randomize don't care values

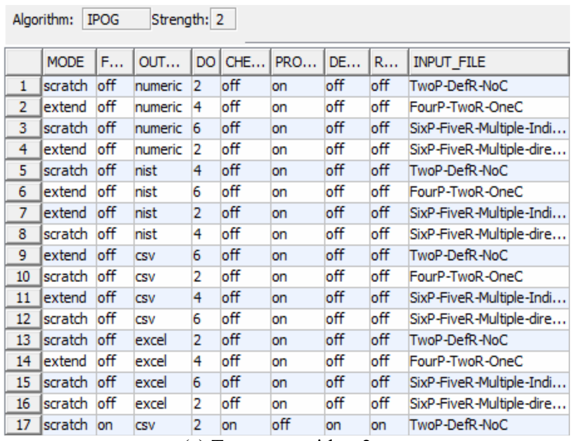
Some of these options e.g. M_fastmode, M_check, M_debug, M_randstar, and M_progressare are totally independent from each other. Because there is no interaction between them they must appear in the test only once. Figure 1 (a) shows the test cases generated by ACTS for the command line interface with test strength t=2. We extended it to t=3 to see whether we could detect more faults. Figure 1(b) shows some of the test cases generated by ACTS for t=3.

C. Graphical User Interface Modeling

ACTS is a complex system with several features and functionalities. The divide-and-conquer strategy is used to model the GUI. We divided the system based on the system use-cases.

The use-cases are often used to capture the system functionalities. We derived the ACTS's use-cases from the user document and captured several features for the GUI such as Create New System, Building the Test Set, Modify system (add/remove/edit parameters and parameters values, add/remove relations, add/remove constraints), Open/Save/Close System, Import/Export test set, statistics, and Verify Coverage.

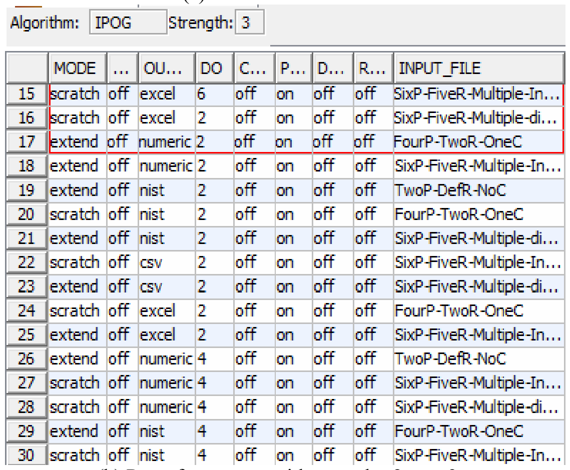
- For each of these we designed a separate IPM to yield several small IPMs rather than one large one. Some of the IPMs have been reported in this paper and for the purpose of brevity others will be reported in the appendix.
- ***Modify system*****:** Modification is the process of changing the system configuration. Designing the IPM for this feature was very challenging because this feature has several functionalities. We divided the modification to the following smaller IPMs; add parameter, remove parameter, modify parameter, add constraint, remove constraint, add relation, and remove relation. Modify a parameter by itself consists of three IPMs; change the name of the parameter, add new value to the parameter, and delete a value from a parameter. In the following we explain some of these models.



Algorithm: IPOG Strength: 2

	MODE	F...	OUT...	DO	CHE...	PRO...	DE...	R...	INPUT_FILE
1	scratch	off	numeric	2	off	on	off	off	TwoP-DefR-NoC
2	extend	off	numeric	4	off	on	off	off	FourP-TwoR-OneC
3	scratch	off	numeric	6	off	on	off	off	SixP-FiveR-Multiple-Indi...
4	extend	off	numeric	2	off	on	off	off	SixP-FiveR-Multiple-dire...
5	scratch	off	nist	4	off	on	off	off	TwoP-DefR-NoC
6	extend	off	nist	6	off	on	off	off	FourP-TwoR-OneC
7	extend	off	nist	2	off	on	off	off	SixP-FiveR-Multiple-Indi...
8	scratch	off	nist	4	off	on	off	off	SixP-FiveR-Multiple-dire...
9	extend	off	csv	6	off	on	off	off	TwoP-DefR-NoC
10	scratch	off	csv	2	off	on	off	off	FourP-TwoR-OneC
11	extend	off	csv	4	off	on	off	off	SixP-FiveR-Multiple-Indi...
12	scratch	off	csv	6	off	on	off	off	SixP-FiveR-Multiple-dire...
13	scratch	off	excel	2	off	on	off	off	TwoP-DefR-NoC
14	extend	off	excel	4	off	on	off	off	FourP-TwoR-OneC
15	scratch	off	excel	6	off	on	off	off	SixP-FiveR-Multiple-Indi...
16	scratch	off	excel	2	off	on	off	off	SixP-FiveR-Multiple-dire...
17	scratch	on	csv	2	on	off	on	on	TwoP-DefR-NoC

(a) Test cases with t=2



Algorithm: IPOG Strength: 3

	MODE	...	OU...	DO	C...	P...	D...	R...	INPUT_FILE
15	scratch	off	excel	6	off	on	off	off	SixP-FiveR-Multiple-In...
16	scratch	off	excel	2	off	on	off	off	SixP-FiveR-Multiple-di...
17	extend	off	numeric	2	off	on	off	off	FourP-TwoR-OneC
18	extend	off	numeric	2	off	on	off	off	SixP-FiveR-Multiple-In...
19	extend	off	nist	2	off	on	off	off	TwoP-DefR-NoC
20	scratch	off	nist	2	off	on	off	off	FourP-TwoR-OneC
21	extend	off	nist	2	off	on	off	off	SixP-FiveR-Multiple-di...
22	scratch	off	csv	2	off	on	off	off	SixP-FiveR-Multiple-In...
23	extend	off	csv	2	off	on	off	off	SixP-FiveR-Multiple-di...
24	scratch	off	excel	2	off	on	off	off	FourP-TwoR-OneC
25	extend	off	excel	2	off	on	off	off	SixP-FiveR-Multiple-In...
26	extend	off	numeric	4	off	on	off	off	TwoP-DefR-NoC
27	scratch	off	numeric	4	off	on	off	off	SixP-FiveR-Multiple-In...
28	scratch	off	numeric	4	off	on	off	off	SixP-FiveR-Multiple-di...
29	extend	off	nist	4	off	on	off	off	FourP-TwoR-OneC
30	scratch	off	nist	4	off	on	off	off	SixP-FiveR-Multiple-In...

(b) Part of test cases with extend t=2 to t=3

Figure 1: CMD test cases created by ACTS

•

- ***Add a parameter*****:** First, adding a parameter; user has to enter a parameter name to activate the add button. We call this M_name in the model with [valid, invalid] test values. The user may enter space or a special character or number but these are invalid and the system will show the related error messages.

The only acceptable name is string without any space. Next selecting a type (M_type) and entering value for the parameter (M_value); also these parameters can be input parameters or output (M_in_out). In addition if the type of the parameter is "Boolean" then the user cannot enter any value because the system has a default value for the Boolean types [true,false], also if type is "Range" the user cannot enter any value but selecting the range. This range could be an invalid range:

(-9000000 to 9000000)

Basically these are some invalid combinations between the type and the value which we have to exclude from the final test cases. As we mentioned before, ACTS has a constraint support feature so we will add the following constraints:

```
M_type="Boolean" => M_value="Default"
```

This means that if the type of the parameter is a "Boolean", the system will fix the value to "default".

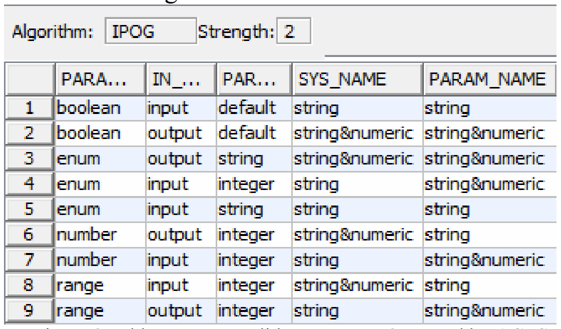
The model of "add parameter" is shown in Table 14. The valid test cases generated by ACTS with test strength t=2 are shown in Figure 2.

Table 14: GUI, add parameter IPM

IPM of GUI, add parameter	
Test Factors	**Test Values**
M_sys_name	invalid (space, special_char, number, duplicate name)
	String only
	String plus numeric
M_name	invalid (space, special_char, number, duplicate name)
	String only
	String plus numeric
M_type	Boolean
	Enum
	number
	range
M_in_out	input
	Output
M_value	Integer
	String
	default
	Invalid (Space, duplicate value, invalid range of numbers or characters)

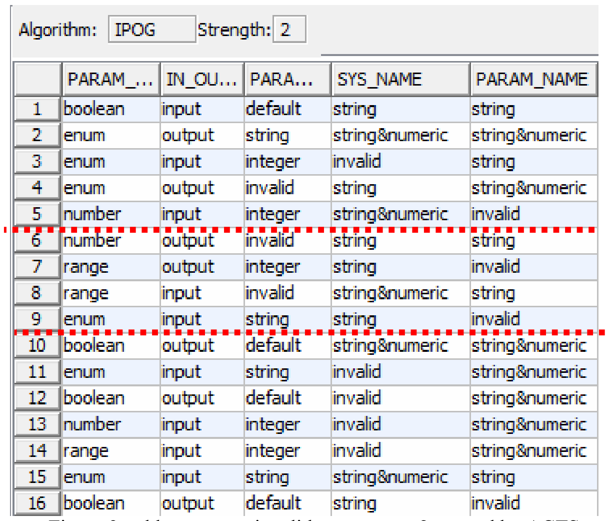
The Invalid test cases generated by ACTS with strength t=2 are shown in Figure 3.

• ***Change parameter name***: The user can change the name of a parameter. The new name should be a valid name (no space). This parameter should also not be involved in any constraint, otherwise the name has to change automatically everywhere in the system in which this parameter is used. The model of "change parameter name" is shown in Table 15. The valid test cases generated by ACTS with strength t=2 are shown in Figure 4.


Algorithm: IPOG Strength: 2

	PARA...	IN_...	PAR...	SYS_NAME	PARAM_NAME
1	boolean	input	default	string	string
2	boolean	output	default	string&numeric	string&numeric
3	enum	output	string	string&numeric	string&numeric
4	enum	input	integer	string	string&numeric
5	enum	input	string	string	string
6	number	output	integer	string&numeric	string
7	number	input	integer	string	string&numeric
8	range	input	integer	string&numeric	string
9	range	output	integer	string	string&numeric

Figure 2: add parameter valid test cases t=2 created by ACTS


Algorithm: IPOG Strength: 2

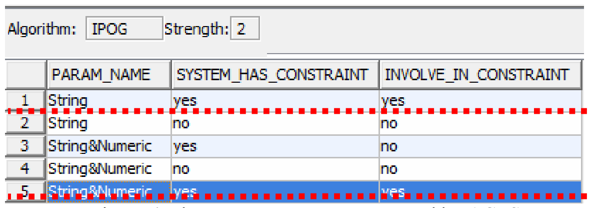
	PARAM_...	IN_OU...	PARA...	SYS_NAME	PARAM_NAME
1	boolean	input	default	string	string
2	enum	output	string	string&numeric	string&numeric
3	enum	input	integer	invalid	string
4	enum	output	invalid	string	string&numeric
5	number	input	integer	string&numeric	invalid
6	number	output	invalid	string	string
7	range	output	integer	string	invalid
8	range	input	invalid	string&numeric	string
9	enum	input	string	string	invalid
10	boolean	output	default	string&numeric	string&numeric
11	enum	input	string	invalid	string&numeric
12	boolean	output	default	invalid	string&numeric
13	number	input	integer	invalid	string&numeric
14	range	input	integer	invalid	string&numeric
15	enum	input	string	string&numeric	string
16	boolean	output	default	string	invalid

Figure 3: add parameter invalid test cases t=2 created by ACTS

M_Involve_in_constraint is a factor to guarantee we will test parameters that are involved in the constraints if the system has a constraint.

Table 15: GUI, change parameter name IPM

IPM of GUI, change parameter name	
Test Factors	**Test Values**
M_name	String only
	String plus numeric
	Invalid (space, special_char, number, duplicate name)
M_Involve_in_constraint	yes
	no
M_System_has_constraint	yes
	no


Algorithm: IPOG Strength: 2

	PARAM_NAME	SYSTEM_HAS_CONSTRAINT	INVOLVE_IN_CONSTRAINT
1	String	yes	yes
2	String	no	no
3	String&Numeric	yes	no
4	String&Numeric	no	no
5	String&Numeric	yes	yes

Figure 4: change parameter name created by ACTS

• ***Building system***: The IPM of build system is shown in Table 16 and Table 17. All of the parameters are discussed earlier because these are core features of ACTS. Valid IPM is used to test the normal functionality of the system and invalid IPM is used for robustness testing.

In the above paragraphs we discussed how we created our models and used them as an input to ACTS tool, and how ACTS give us all the combinations between factors for each model. The number of models was 19 with 1105 generated test cases.

We integrated the smaller IPMs together using an interaction-based test sequence generation to completely test the system. The reason we decided to use this method was that some of the bugs would not be triggered by just testing each use-case individually. It is important to test a sequence of events in order to test the whole system completely. Wenhua Wang et al. [14] present a test sequence generation approach for covering all interactions between any two pages of a web application.

Table 16: GUI, Build valid IPM

Valid IPM of GUI, build system	
Test Factors	**Test Values**
I mode	scratch
	extend
I algorithm	ipog
M_Strength	2,4,6
M_randomize	on
	off
M_progress	on
	off
M_SUT	2P_2R_multi-nC
	multiP_multiR_multi-rC
	multiP_multiR_oneC
	2P_2R_noC

Table 17: GUI, Build invalid IPM

Invalid IPM of GUI, build system	
Test Factors	**Test Values**
I mode	scratch
	extend
I algorithm	ipog
M_Strength	2,4,6
M_randomize	on
	off
M_progress	on
	off
M_SUT	InvalidP
	InvalidR
	InvalidC
	UnsolvedC

We can generalize this algorithm to be able to use it in combinatorial testing of systems with a GUI as well. First we generated a navigation graph of our use cases. There exists an edge from one node m to another node n if node n can be visited immediately after node m through a direct link. Each node is a use-case. A simplified form of ACTS's use-cases navigation graph is shown in Figure 5. Using the navigation graph was very helpful because not all the combinations between the use-cases are feasible. The graph helped to visualize the feasible and infeasible sequences.

Next we generated a test sequence to satisfy pairwise interaction coverage. The term "pairwise interaction" refers to interaction between two nodes. Let G = (V, E, n0) be a navigation graph. Formally, a pairwise interaction in G is an ordered pair (m, n), where m and n are two nodes, and there exists a path from m to n in G. Pairwise interaction coverage requires that a set of paths be selected from a navigation graph as test sequences so that every ordered pair is covered in at least one of those test sequences. We generated all of the ordered pairs for use-cases from the navigation graph.

In next Section we provide some examples of the sequences that lead us to find the faults in ACTS. In this experiment we limited the length of sequences to be six. The whole process, from generating the graph, to selecting the proper interactions, to selecting the sequence of events, was performed manually. In this paper our focus was on IPM for ACTS. In future work we will use tools such as *GUI Ripper* to remove human error in this part of the experiment [28].

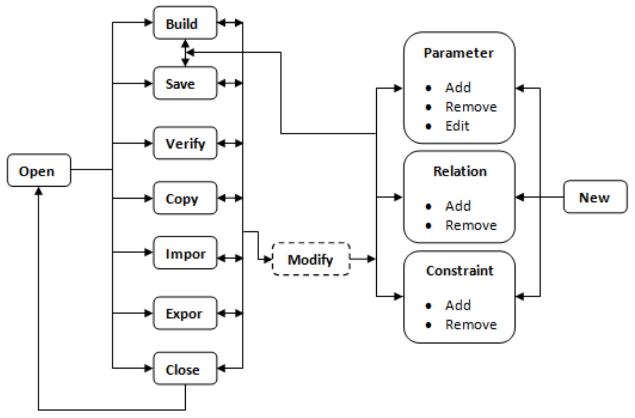


Figure 5: ACTS's Navigation Graph

V. Experimental Results

The experiments are designed to answer the following questions: How much code coverage can be achieved? How many faults can be detected?

The design model for ACTS has 19 valid IPMs which are shown in Table 18, yielding 1105 generated test cases. Code coverage data are shown in Figure 6 and Figure 7. We used *clover* to collect code coverage [15]. We ran *clover* with eclipse and executed our tests on ACTS version 1.2. ACTS statistics are shown in Table 19. e.g. number of uncommented lines of code in ACTS are 24637.

Table 18: IPM of ACTS

Model	Number of Factors	Max number of values
CMD	7	8
BUILD	6	8
NEW SYSTEM	4	5
ADD PARAM	4	5
REMOVE PARAM	2	2
CHANGE NAME	2	2
ADD VALUE	3	5
REMOVE VALUE	3	3
ADD RELATION	2	4
REMOVE RELATION	2	4
ADD CONSTRAINT	3	3
REMOVE CONSTRAIN	2	5
OPEN	3	4
CLOSE	2	2
VERIFY	2	2
IMPORT	3	4
EXPORT	2	4
SAVE	3	2
STATISTICS	3	3

Clover gave us the code coverage for all of the test cases. While we executed our tests, clover highlighted the parts of the source code that were executed. This made it easy to identify the code that was never called during our testing process. It is shown in Figure 6 that our tests covered more than 88% of system statements. Figure 7 shows different packages of ACTS. We covered 99% the Console package.

Other packages are more related to the GUI. Packages, e.g. Engine, Model, Util, GUI, and Data are common between different algorithms. We only performed testing on the "IPOG" algorithm. There are five more algorithms implemented in ACTS. Therefore, we have not exercised some statements in our experiments.

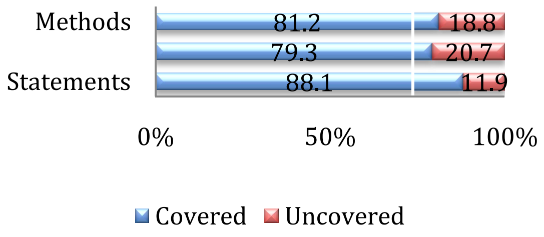


Figure 6: ACTS Effectiveness Metrics

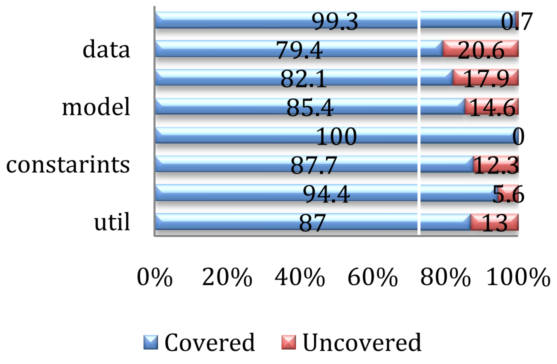


Figure 7: Statement coverage for ACTS packages

We classified detected faults in ACTS into four groups as shown in Table 20. The First group is the faults related to functionality testing of graphical user interface. The second group is the faults related to robustness testing of graphical user interface. The third group is the faults related to functionality testing of command line interface. The fourth group is the faults related to robustness testing of command line interface.

Table 19: ACTS Statistics

LOC (line of code)	38,165
NC LOC	24,637
Number of Statements	13,642
Number of Branches	4,696
Number of Methods	1,693
Number of Classes	153
Number of Files	110
Number of Packages	12

The total number of detected faults is 15, 10 of which detected by functionality testing and 5 of them detected by robustness testing. In our experiment some of the faults detected in the GUI occurred in the command line interface as well. One possible reason that we only detect 15 faults out of almost 1000 tests is because the ACTS is pretty mature software, well documented, stable, and widely used, Also some of the detected bugs are single mode faults.

The following are some examples of the detected faults. The red lines in Figure 3 show the test cases that detect bugs. For example, the first line is a bug with the scenario that system let the user enter a space character, which is an invalid value for the Enum type. The second red line is another bug with the scenario that the system let the user select an invalid range for the Range types. Both of these bugs are detected during robustness testing of GUI. The red line in Figure 4 also shows another detected bug with the scenario that the system lets the user change the name of the parameter that is involved in the constraints.

Table 20: Faults Classification

Fault Groups	Number of Detected Faults
functionality testing of GUI	10
robustness testing of GUI	5
functionality testing of cmd	1
robustness testing of cmd	1

The following are two examples of the detected faults of the sequences that lead us to find the fault in ACTS. Assume L is our node list. L1 and L2 are two different test sequences that led us to detect three different bugs in ACTS.

- *L1 = {open, import, build (extend mode), save, close}*

In this scenario, the user opens a system, imports the test set and builds it. An error was detected when we built the system in this scenario. The imported test set had an invalid format, which caused the build process to throw an exception. This error was not detected by functional testing of the import use-case individually. The import method failed to correctly set all the values that are needed by the back-end system parameters. However, this problem was not observed from outside when we tested the import use-case. The build operation after the import operation helped to expose the incorrect state as an exception that can be observed from outside.

- *L2 = {open, build, Edit a parameter, build (extend mode)}*

In this scenario, the user opens a system, builds the system, and edits a parameter. The Edit operation, as explained in "modify parameter" section, allows values of a parameter to be added or removed. After modifying a parameter, the user builds the system again. An error was detected after we called the build method again, this time in the extend mode. This error was not detected by testing the "modify parameter" use-case individually. Similar to L1, the modify method failed to correctly set all of the values to the back-end system parameters. This problem was, however, only be exposed when we built the system again.

VI. THREATS TO VALIDITY

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. We have tried to automate the experimental procedure as much as possible, in an effort to remove human errors. We generate our test cases with ACTS, which is automated, and we executed them on the command interface of ACTS automatically. We used clover, which is a third party application to measure our code coverage.

Threats to external validity occur when the experimental results could not be generalized to other programs. We used ACTS, which is only one application. More experiments on other programs can improve the external validity of our study.

VII. Conclusion and Future Work

This paper presents a case study on applying combinatorial testing to test a combinatorial test generation tool called ACTS. The main challenge of this study was modeling the input space of ACTS in terms of a set of parameters and values. In particular, significant effort was spent on modeling the System Under Test (SUT), which may have different types of parameters, relations and constraints, and on modeling the GUI interface, for which several smaller models were created and tested and then integrated together. The results of this study indicate that input space modeling is a significant task, and it needs to be managed carefully. The results of this study also show that combinatorial testing is effective in terms of achieving high code coverage and fault detection.

We plan to conduct similar studies of other real-world applications. The goal is to develop a set of guidelines, with significant examples, that can be used by practitioners to apply combinatorial testing in practice.

VIII. Acknowledgment

This work is supported by two grants (70NANB9H9178 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST) and a grant (61070013) of National Natural Science Foundation of China.

IX. Refrences

[1]. Grindal, M. and Offutt, J. (2007) Input Parameter Modeling for Combination Strategies in Software Testing, Proceedings of the IASTED International Conference on Software Engineering (SE2007), Innsbruck, Austria, 13-15 Feb 2007, pages 255-260

[2]. Grindal, M., Offutt, J. and Mellin, J. (to appear) Managing Conflicts when Using Combination Strategies to Test Software, Proceedings of the 18th Australian Conference on Software Engineering (ASWEC2007), Melbourne, Australia, 10-13 April 2007.

[3]. Grindal, M, Offutt, J, and Andler, S. F. (2005) Combination Testing Strategies: (A) Survey, publisher Wiley, Software Testing, Verification, and Reliability, volume 15, number 2, pp. 167-199.

[4]. Grindal, M. (2007) Handling Combinatorial Explosion in Software Testing. Thesis Dissertation no 1073, Department of Computer and Information Science Linköpings universitet. ISBN 978-91-87515-74-9. ISSN 0345-7524

[5]. T. Chen, P.-L. Poon, S.-F. Tang, and T. Tse. On the Identification of Categories and Choices for Specification-based Test Case Generation. Information and Software Technology, 46(13):887–898, 2004.

[6]. M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. Journal of Software Testing, Verification, and Reliability, 3(2):63–82, 1993.

[7]. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM, 31(6):676–686, June 1988.

[8]. http://www.pjcj.net/testing_and_code_coverage/paper.html by Paul Johnson

[9]. T. Chen, S.-F. Tang, P.-L. Poon, and T. Tse. Identification of Categories and Choices in Activity Diagrams. In Proceedings of the Fifth International Conference on Quality Software (QSIC 2005) 19-20 September 2005, Melbourne, Australia, pages 55–63. IEEE Computer Society, September 2005.

[10]. B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, 1990.

[11]. Y. K. Malaiya. Antirandom testing: Getting the most out of black-box testing. In Proceedings of the International Symposium on Software Reliability Engineering, (ISSRE'95), Toulouse, France, Oct, 1995, pages 86–95, Oct. 1995.

[12]. H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. Automatic Test Generation using Checkpoint Encoding and Anti-random Testing. Technical Report CS-97- 116, Colorado State University, 1997.

[13]. D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The Combinatorial Design Approach to Automatic Test Generation. IEEE Software, 13(5):83–89, September 1996.

[14]. Wenhua Wang, Sreedevi Sampath, Yu Lei, Raghu Kacker. An Interaction-Based Test Sequence Generation Approach for Testing Web Applications, IEEE International Conference on High Assurance Systems Engineerng, December 2008.

[15]. Clover: Code Coverage Tool for Java. http://www.cenqua.com/clover/.

[16]. Kuhn R, Wallace D, Gallo A. Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 2004; 30(6):418–421.

[17]. C. Nie and H. Leung. A survey of combinatorial testing. ACM Computing Surveys (CSUR), 43:11:1–11:29, 201

[18]. Krishnan, R.,Krishna, S. M., Nandhan, P. S. 2007. Combinatorial Testing: Learnings From Our Experience. Sigsoft Softw. Engin. Notes 32, 3, 1–8.

[19]. Burr, K. and Young, W. 1998. Combinatorial Test Techniques: Table -based Automation, Test Generation, And Code Coverage. In Proceedings Of The International Conference On Software Testing Analysis And Review. 503–513.

[20]. Lei, Y., Carver, R. H., Kacker, R., and Kung, D. C. 2007. A Combinatorial Testing Strategy for Concurrent Programs. Softw. Test., Verif. Reliab. 17, 4, 207–225.

[21]. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html

[22]. Huller, J. 2000. Reducing Time to Market with Combinatorial Design Method Testing. In Proceedings of The The International Council On Systems Engineering (Incose) Conference.

[23]. Xu, B., Nie, C., Shi, L., Chu, W. C., Yang, H., and Chen, H. 2003. Test Plan Design for Software Configuration Testing. In Software Engineering Research and Practice, Csrea Press, 686–692.

[24]. Xu, L., Xu, B., Nie, C., Chen, H., and Yang, H. 2003b. A Browser Compatibility Testing Method Based On Combinatorial Testing. In Proceedings of the International Conference on Web Engineering (Icwe. Springer, Berlin, 310–313.

[25]. Williams, A. W. And Probert, R. L. 1996. A Practical Strategy for Testing Pair-wise Coverage of Network Interfaces. In Proceedings of the 7th International Symposium on Software Reliability Engineering (Issre'96). Ieee Computer Society, Los Alamtos, Ca, 246.

[26]. Burroughs, K., Jain, A., and Erickson, R. 1994. Improved Quality Of Protocol Testing Through Techniques Of Experimental Design. In Proceedings of the IEEE International Conference on Record, 'serving Humanity through Communications.' Vol. 2. 745–752.

[27]. A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," IEEE Transactions on Software Engineering, vol. 31, no. 10, pp. 884–896, 2005.