An Input Space Modeling Methodology for Combinatorial Testing

Mehra N.Borazjany, Laleh Sh. Ghandehari, Yu Lei
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019, USA
{mehra.nourozborazjany}@uta.edu
{laleh.shikhgholamhosseing,ylei }@uta.edu

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

Abstract— **The input space of a system must be modeled before combinatorial testing can be applied to this system. The effectiveness of combinatorial testing to a large extent depends on the quality of the input space model. In this paper we introduce an input space modeling methodology for combinatorial testing. The main idea is to consider the process of input space modeling as two steps, namely, input structure modeling and input parameter modeling. The first step tries to capture the structural relationship among different components in the input space. The second step tries to identify parameters, values, relations and constraints for individual components. We also suggest strategies about how to perform unit and integration testing based on the input space structure. We applied the proposed methodology to two real-life programs. We report our experience and results that demonstrate the effectiveness of the proposed methodology.**

***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 [8]. Combinatorial testing 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 the value combinations of t (out of k) parameters be covered at least once, where t is usually a small integer. If test parameters are modeled properly, almost all faults caused by interactions involving no more than t parameters will be exposed. Combinatorial testing can significantly reduce the cost of software testing while increasing its effectiveness [30,31].

The input space of a system must be modeled before combinatorial testing can be applied to this system. The effectiveness of combinatorial testing to a large extent depends on the quality of the input space model. In particular, if a failure can only be triggered when a parameter takes a specific value, and if this parameter or value is not modeled, this failure will not be detected by a combinatorial test set.

A number of studies have been reported on input space modeling for general software testing, i.e., not specific to combinatorial testing. Grochtmann and Grimm [3] mentioned that finding parameters and values is a creative process that can never be fully automated. The Category Partition method partitions the input domain into categories and choices [4]. The Classification Tree method partitions the input domain into classifications and classes, and represents the input domain in a tree structure [3]. Note that classifications are like categories and classes are like the choices.

Only a few studies have been reported on modeling for combinatorial testing. A workflow of eight steps is proposed for the combinatorial modeling process [1]. Segall et al. suggest several patterns i.e., optional values, multiplicity that commonly appear in input models for combinatorial testing [27, 28].

In this paper we propose an input space modeling strategy for combinatorial testing. We consider the process of input space modeling as two steps: input structure modeling (ISM) and input parameter modeling (IPM). The first step, i.e., ISM, tries to capture the structural relationship among the different components in the input space. The second step, i.e., IPM, tries to identify parameters, values, relations and constraints for individual components. We also suggest strategies about how to perform unit and integration testing based on the input space structure.

In this paper, we focus on the first step. Existing methods such as category partitioning can be used for the second step. We consider two types of structures, i.e., flat and graph. The flat structure has no compositional hierarchy, where components are equal peers in terms of composition relation. For example, a flat input structure can be used to model the command-line options of a program. The graph structure represents the composition relation between different components in a graph, where one component may be composed of several other components. For example, a graph structure can be used to model elements in an XML file.

We also report two experiments of applying the proposed methodology to two real-life programs, including *Apache Ant* [17], and *Space* from SIR website [19]. The two experiments are designed to serve two purposes. First, they are designed to validate the proposed methodology in a practical setting. Second, they are intended to evaluate the effectiveness of combinatorial testing. There has been a lack

of empirical studies and experience reports on applying combinatorial testing to the real-life programs [9].

In our experiments, we compare combinatorial testing based on the proposed methodology to two other random approaches. The first random approach, referred to as pure random, spends minimum effort on modeling and generates random tests mainly based on the syntactic structure of the input space. The second random approach, referred to as modeled-random, generates random tests from the same model created by the proposed methodology. We measure the effectiveness of these approaches in terms of code coverage and number of faults they can detect. The results show that our approach is more effective than the modeled-random approach, which is significantly more effective than the pure-random approach. For example, the statement coverage in *Apache Ant* with 2121 test cases was 79% using our approach as compared to modeled-random and pure-random approaches with 65% and 44% respectively.

The remainder of this paper is organized as follows. In Section II we discuss existing work on input space modeling. In section III, we describe our methodology for input space modeling. Section IV reports experiments that demonstrate the effectiveness of the methodology. Section VI provides concluding and future works.

II. Related Work

Several input space modeling approaches, e.g., Category Partitioning [4], Classification Tree [3], and Domain Testing [5] have been reported for general software testing. The general idea is to divide the possible values of a parameter into groups that result in a similar behavior from the test subject.

Chen et al. [2] presented a list of common mistakes in identification of the category (parameter) and choices (values) from specification. The missing categories, problematic categories, and problematic choices are the main mistakes. They introduced a check list of 6 steps for detecting these mistakes.

Grindal and Offutt suggested an input parameter modeling method that is specifically designed for combinatorial testing [1]. The focuses of this study was mainly on work flow of the modeling process.

Segall et al. [27, 28] and Lott et al. [26] studied some patterns that commonly occur in combinatorial test models. Common patterns include optional values, multiplicity, and auxiliary aggregates or commonality. The concept of reusing commonality or auxiliary aggregates [26] is different from our approach as discussed later in section III-C.

The notion of sub-attribute introduced in [10] allows one factor to be composed of several other factors. This is similar to our graph structure. However, the approach in [10] splits the composed factor into simple factors during test generation. This effectively converts the graph structure into a flat structure, which is fundamentally different from our approach.

To complement mixed-strength test generation, a user is allowed to create a hierarchy of test parameters [43]. This is similar to our graph structure without loop. However, the hierarchy in [43] can have two levels only while our approach does not have this restriction. Moreover our approach deals with loops in a graph structure.

The above works are complementary to our work. None of the above addresses the problem of input structure modeling.

A number of empirical studies have been reported on combinatorial testing. In [8], Kuhn et al. reported a study of several fault databases and found that all the faults in these databases are caused by no more than 6 factors. Schroeder et al. [23] compared t-way testing to random testing in a controlled study. They used two software applications used in their laboratory as subject programs and seeded faults by themselves. Qu et al. [35] applied combinatorial testing to two programs in the SIR repository, i.e., flex and make. Kuhn et al. [21] applied t-way testing to a grid network simulator for deadlock detection and compared the results to random testing. These studies are complementary to our work in that they all provide evidence and insights on the effectiveness of combinatorial testing in practice. However, none of these studies reported the details of the modeling process.

III. An Input Sapce Modeling Methodology

For a large system that has too many features, we first use the divide-and-conquer strategy to divide the system into smaller systems. There are two general strategies. One is to divide the system vertically, e.g., based on features. That is, we could apply combinatorial testing to one feature or a group of related features at a time. The other strategy is to divide the entire system into several subsystems, where each subsystem may be involved in multiple features.

Next, we model the input space of each system. This modeling process consists of two major steps, Input Structure Modeling (ISM) and Input Parameter Modeling (IPM). ISM tries to capture the structural relationship among the different components in the input space. We consider two types of structures, i.e., flat and graph. The flat structure has no hierarchy, where components are equal peers. For example, a flat input structure can be used to model the command-line options of a program. The graph structure represents the composition relation between different components in a graph, where one component may be composed of several other components. For example, a graph structure can be used to model elements in an XML file. In this study, we focus on the graph structure.

In [10], the graph structure is referred to as sub-attributes and it was suggested to either consider the parent node as a compound parameter or split the parent node into simple parameters. In their experiments they used the split approach because it was believed that this approach would generate a less number of test cases. The split approach effectively converts the graph structure into a flat structure, which creates more parameters and may also introduce many

invalid combinations. The split approach may also introduce redundant factors if one child has more than one parent.

IPM tries to identify parameters, values, relations and constraints for individual components. Existing methods on IPM can be applied. In particular, it is often necessary to identify abstract parameter and values. One common approach is to identify factors that could affect the behavior of the object being modeled. Each of these factors can become an abstract parameter. Then, existing methods such as category partitioning and classification tree can be used to identify the abstract values of each abstract parameter.

After we have the input parameter model for each module, we generate test cases from the model using combinatorial testing tools such as ACTS. These test cases are abstract test cases because the parameters and values in the model are abstract. Thus, it is necessary to derive concrete test cases from these abstract test cases before the actual testing can be performed. Note that an abstract test case typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing.

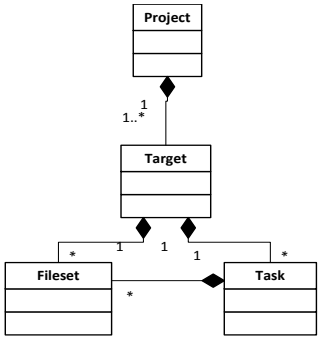
A. Input Structure Modeling

In this section, we focus on the graph structure. We consider two types of graph structure, depending on whether there is a loop in the graph structure.

Fig.1: (a) part of a build XML file used *by Apache Ant*. (without loop)

```
<project name="helloworld" basedir=".">
   <target name="compile">
        <mkdir dir="classes"/>
        <fileset file="main.java"/>
        <javac srcdir="src" destdir="classes"/>
   </target>
   <target name="jar">
        <mkdir dir="jar"/>
        <jar destfile="jar/HelloWorld.jar">
             <fileset dir="classes"/>
        </jar>
   </target>
</project>
```

(b) The model of the example build file



i. Graph structure without loop

As an example, consider the build XML file used by *Apache Ant*, which is shown below. A build XML file includes one project element and at least one target element. A target element includes one or more task elements. Each element has some attributes.

Fig.1 (a) shows part of a build XML file used *by Apache Ant*. The file contains the 'helloworld' project which has two targets. The first target 'compile' has two task elements, 'mkdir' and 'javac', and one 'fileset' element. The second target 'jar' has two task elements, 'mkdir' and 'jar'. The task element 'jar' has a nested 'fileset' element.

Fig.1 (b) shows the model of the example build file in Fig.1 (a). We borrow some UML notations to depict the graph structure. These UML notations include class, attribute, multiplicity, constraint and composition relation notations.

The multiplicity notation shows that a project element has at least one target element. The composition relationship shows that the target and the task both use the fileset element. This suggests that we can model the fileset element once and then reuse it for both task and target elements. Note that the input structure is modeled as a graph even though the structure of the xml input file is a tree structure.

In the following, we give some guidelines on how to perform unit testing and integration testing based on a graph structure.

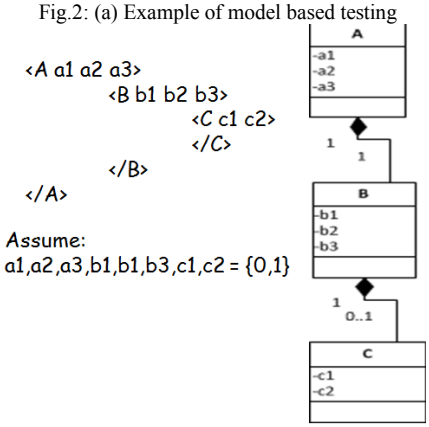
Fig.2 (a) shows an example of a graph structure. Nodes A, B, and C are the parameters of the system and a1, a2, and a3 are the attributes of the node A and so on.

- **Unit testing:** Unit testing can be performed to test different combinations of attributes for individual nodes. In this case, attributes for the other required nodes will be given a default value. If an attribute has no default value, we will either exclude this attribute or will pick a value that we consider may be used most often. Fig.2 (b) shows an example of how to perform unit testing for the structure shown in Fig.2 (a).
- **Integration testing:** Integration testing can be performed after unit testing of each node. The child node will be used as a composed parameter of the parent node.

Fig.2 (c) and Fig.2 (d) show two coverage options for the integration strategy. In both examples, the node C is the first node to test. It has two binary attributes c1 and c2 as its parameters. The 2-way test set for node C has four tests. The second node to test is node B. Node B has three attributes with two values (binary attributes) and one nested element C. When we test node B, node C is also considered to be a new parameter of B, whose domain is the two-way test set derived earlier for unit testing of node C.

We have two coverage options for parameter C depending on our domain knowledge. If there is no relationship between the attributes of the two nodes (B and C), then we do one-way coverage between the parameter C and the attributes of node B as shown in Fig.2 (c). The ACTS tool has a mixed relation feature that we can use to generate the mixed coverage between parameters. Otherwise, we perform t-way testing for all the parameters of node B, i.e., including its attributes and the new parameter added for node C as shown in Fig.2 (d).

Fig.2: (a) Example of model based testing



B		
b1	b2	b3
1	1	0
1	0	1
0	1	1
0	0	0

A		
a1	a2	a3
1	0	1
1	1	0
0	0	0
0	1	1

C	
c1	c2
0	0
0	1
1	0
1	1

(b) Unit testing

C	
c1	c2
0	0
0	1
1	0
1	1

B			
b1	b2	b3	C
1	1	0	1
1	0	1	2
0	1	1	3
0	0	0	4

A			
a1	a2	a3	B
1	0	1	1
1	1	0	2
0	0	0	3
0	1	1	4

(c) Integration testing (mixed-coverage)

C	
c1	c2
0	0
0	1
1	0
1	1

B			
b1	b2	b3	C
0	1	1	1
1	0	0	1
0	0	1	2
1	1	0	2
0	0	0	3
1	1	1	3
0	0	0	4
1	1	1	4

A			
a1	a2	a3	B
0	1	1	1
1	0	0	1
0	0	1	2
1	1	0	2
0	0	0	3
1	1	1	3
0	0	0	4
1	1	1	4
0	0	0	5
…	…	…	..

(d) Integration testing (t-way coverage)

ii. Graph structure with loop

Some graph structures may contain a loop. Fig.3 shows an example where there exists a loop involving nodes ‘classpath’, ‘fileset’, ‘depend’, and ‘mapper’. (The edges are directed in terms of the compositional relation.)

In the modeling process:

(1) Break the loop by removing the back edge. This is necessary to determine the order in which the nodes are going to be modeled. The back edge points from a node to another node that has already been visited during a DFS traverse.

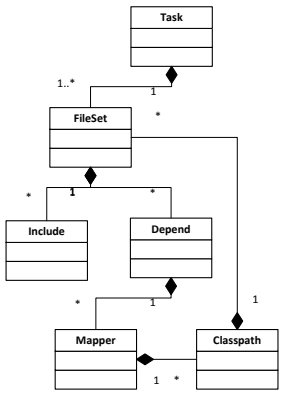
(2) Model the graph without the loop (as we discussed in previous section).

(3) Place the back edge into the graph and add the destination node of the back edge into the model of the source node of the back edge.

Fig.3: (a) Part of build.xml file used by *Apache Ant* (with a loop)

```
<fileset dir="src" includes="*.java">
    <depend targetdir="/lib">
        <mapper classname="mapper.classname">
            <classpath>
                <fileset dir="lib">
                    <include name="*.class"/>
                </fileset>
            </classpath>
        <mapper>
    </depend>
</fileset>
```

(b) The model of the example build file



In the above example, we break the loop by removing the back edge going from node ‘classpath’ to ‘filset’. After we modeled the graph without loop then we add the back edge and remodel the ‘classpath’ node. The ‘classpath’ node using the ‘fileset’ node as its nested element and we have already modeled the ‘fileset’ node which we can add to the model of ‘classpath’.

Another important task is to generate the concrete test cases using the model. In order to do so we have to unfold the loop based on our coverage criteria e.g., edge pair coverage, prime path coverage, simple round trip coverage. For example if we want to satisfy the simple round trip coverage, we unfold the loop just once.

B. Input Parameter Modeling

After we model the input structure of the system and in order to perform testing, we need to model the input parameter of each node. The IPM method identifies the abstract model for the parameters, values, constraints, and relations from the specification. One common approach is to identify factors that could affect the behavior of the object being modeled. Each of these factors can become an abstract parameter. The existing methods such as category partitioning and classification tree can be used to identify the

	COMPRESS	FILEONLY	MANIFEST	UPDATE	WHENMANIFESTONLY	DUPLICATE	FILESETMANIFEST	KEEPCOMPRESSION	NESTEDFILESET
1	false	false	currentdir	false	create	preserve	merge	false	one
2	true	true	currentdir	true	fail	fail	merge	true	two or more
3	false	true	currentdir	false	skip	add	merge	true	None
4	true	false	currentdir	true	create	fail	mergewithoutmain	false	None
5	false	true	currentdir	true	fail	add	mergewithoutmain	false	one
6	true	false	currentdir	false	skip	preserve	mergewithoutmain	true	two or more
7	false	true	currentdir	true	create	add	skip	false	two or more
8	true	false	currentdir	false	fail	preserve	skip	true	None
9	false	true	NA	false	skip	fail	skip	false	one
10	true	false	currentdir	false	create	add	NA	true	two or more
11	false	true	currentdir	true	fail	preserve	NA	false	None
12	true	false	currentdir	true	skip	fail	NA	true	one
13	true	false	NA	true	skip	add	skip	true	None
14	true	false	NA	false	skip	preserve	skip	true	two or more

Fig.4: <jar> task 2-way abstract test cases

abstract values of each parameter. The constraints are introduced to avoid invalid combinations. The relations are introduced to group parameters that are related to each other so that the different groups can be covered at different strengths.

C. Derive concrete test cases

We use ACTS to generate t-way abstract test cases for each model. These test cases are abstract test cases because the parameters and values in the model are abstract. It is necessary to derive concrete test cases from these abstract test cases before testing is actually performed. Note that an abstract test case typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing.

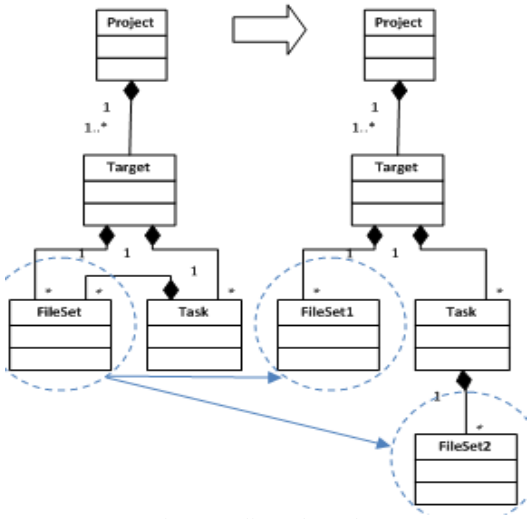


Fig.5: Duplicate the nodes

Since the graph structure represents the composition relation between different components, where one component may be composed of several other components, we often need to duplicate nodes that have more than one parent. Fig.5 shows an example of such duplication. The node 'fileset' has two parents meaning that 'target' and 'task' both will use the 'fileset'. In order to generate the test, we need to duplicate the node 'fileset' to two nodes 'fileset1' and 'fileset2' such that each only has one parent. This does not mean that we model the 'fileset' node two times. Instead we reuse the same abstract model for the 'fileset' node to create two concrete values, which may be the same or different, to be used for the 'task' and 'target' nodes. The concept of auxiliary aggregate is to factor out the common parameters with identical concrete values in the model [26]. In contrast, we reuse the abstract model and the concrete values can be different.

We implemented a test case generator specific to each subject to automatically derive concrete values from the abstract test cases. Fig.4 and Fig.6 show the 2-way abstract test set for the 'fileset' and 'jar' tasks respectively. Each column represents a test factor (or abstract parameter) of the task and each row represents an abstract test case. The test case generator will select one representative for each test case and create a concrete test case.

A sample concrete test for the test case number 14 in the Fig.4 is:

```
<jar destfile="test14.jar" compress="true"
fileonly="false" update="false" duplicate="preseve"
keepcompression="true" >
    <fileset dir="/test" excludes="*.jar"
include="*.class"/>
    <fileset dir="/test" defaultexcludes="yes"
include="*.java" />
</jar>
```

In test case 14, the abstract value of the parameter *nestedfileset* is 'two or more'. Therefore, in the concrete test case we include two nested <fileset> elements. The <fileset> elements are further selected from the <fileset> abstract test cases shown in Fig.6.

	FILE	DIR	DEFAULTEXCLUDES	INCLUDES	INCLUDESFILE	EXCLUDES	EXCLUDESFILE
1	on	off	off	off	off	off	off
2	off	on	off	on	off	on	off
3	off	on	on	off	on	off	on
4	on	off	on	on	off	on	off
5	on	off	off	off	on	on	off
6	off	on	on	on	off	off	off
7	on	off	off	off	on	off	on

Fig.6: <fileset> 2-way abstract test cases

In addition, the test environment should contain a directory name 'test' and files with different extensions such as '.java', '.class' and '.jar', in order for us to test the functionality of the above elements. Therefore, our test generator creates files with predefined extensions inside the 'test' directory.

IV. EXPERIMENTS

We conducted two experiments in which the proposed methodology was applied to two real-life programs: *Apache Ant* and *Space*. The two experiments are designed to serve two purposes. First, they are designed to validate the proposed methodology in a practical setting. Second, they are intended to evaluate the effectiveness of combinatorial testing.

In our experiments, we compared combinatorial testing based on the proposed methodology to two other random approaches. The first random approach, referred to as pure-random, spends minimum effort on modeling and generates random tests mainly based on the syntactic structure of the input space. The second random approach, referred to as modeled-random, generates random tests from the same model created by the proposed methodology.

We also considered the possibility of comparing our approach to an approach where we could create a model purely on the syntactic structure of the input space and then generate a t-way test set from this syntactical model. This approach requires minimal modeling effort, since no semantic information needs to be considered in the modeling process. The two approaches differ only in the modeling process. Thus this comparison would be good to show the difference caused by different modeling approaches. However, we found it was difficult to implement this approach. For example, we tried to apply this approach to the *Apache Ant* program and identified about 300 parameters. In addition, we had to introduce a large number of constraints to reflect the hierarchical structure. Thus we did not perform this comparison in our experiments.

A. Subject programs

Table 1: Statistics about the two subject programs

Subject programs	LOC	# of classes/ procedures	# of Faults in each faulty version	Type of faults
Ant 1.6 beta	80500	627	6	Seeded
Space	9127	136	35	Real

Table 1 shows some statistics about the two subject programs. The two programs are selected because of several desired attributes, including their complex input space, the existence of a clean version and multiple faulty versions, a relatively large number of lines of code, and the availability of their specifications.

B. Input model

Table 2 shows some information about the input models built for each program. The second column shows the number of models created for each program. The 3rd column shows the total number of parameters and their domain size in an exponential format, i.e., $(d_1^{p1} d_2^{p2} d_3^{p3})$, where d_i^{pi} indicates that there are p_i number of parameters with domain size of d_i. Note that the total number of parameters equals to p1+p2+p3$_{...}$. The 5th column represents the total number of constraints and the number of parameters involved in each constraint also in an exponential format, i.e., $(p_1^{c1} p_2^{c2} p_3^{c3})$, where p_i^{ci} indicates that there are c_i constraints with p_i parameters. Note that the total number of constraints is equal to c1+c2+c3+$_{...}$. The 7th column shows the total number of relations and the number of parameters involved in each relation also in an exponential format, i.e., $(p_1^{r1} p_2^{r2} p_3^{r3})$, where p_i^{ri} indicates that there are r_i number of relations with p_i number of parameters. The total number of relations is equal to r1+r2+r3+$_{...}$.

For example, *Space* has 7 models with a total number of 78 parameters. Five parameters have the domain size of 2, 21 parameters have the domain size of 3 and so on.

C. Test generation

We used ACTS (version 2.7) [12] to generate t-way abstract test cases. We started from 2-way testing and then we extended the generated test cases to perform 3-way testing. As the number of test cases increases rapidly as the test strength increases, therefore we did not go beyond 3-way testing.

We generated the modeled-random abstract test cases by using the RANDBETWEEN(m,n) function of Microsoft Excel. The integer numbers (m and n) indicate the first and the last index of the parameter values.

For example, consider a model of three parameters with domain size 4, RANDBETWEEN(1,4) generates a random integer between 1 and 4 for each parameter in a test case (Table 3). We used an IF function to check whether a test case satisfies all the constraints. In this study, we only used valid test cases, i.e., invalid tests were discarded.

Table 2: Input Model

Subject programs	Number of models	Total # of parameter and their domain size	Total # of parameters	Total # of constraint and the number of parameter involve	Total # of constraints	Total # of relations and the number of parameter involve	Total # of relations
ANT	53	$2^{96}3^{21}4^{45}5^{6}6^{2}7^{1}9^{2}$	172	$2^{11}3^{18}4^{13}5^{5}7^{5}$	52	$2^{9}3^{3}4^{1}$	13
SPACE	7	$2^{5}3^{21}4^{46}5^{1}6^{2}7^{1}8^{2}$	78	$2^{21}3^{9}4^{8}5^{4}6^{3}7^{1}8^{2}10^{1}14^{7}18^{2}$	58	$2^{14}3^{4}4^{2}6^{1}$	21

We used the sample xml generator *Oxygen* [25], to generate random XML files for the pure-random approach. *Oxygen* is an XML editor that creates XML documents based on a schema or a DTD file. It accepts a DTD file as input and converts it to a XML schema file (i.e., a XSD file). It generates a user-defined number of random xml files from the schema. We set the number of repetitions and recursive levels to 2.

Table 3: Example of random abstract test case generation

Parameters	P1	P2	P3	valid/invalid
Formula	RAND BETWEEN(1,4)	RAND BETWEEN(1,4)	RAND BETWEEN(1,4)	=IF(AND(A2=B2,B2=C2), "invalid", "valid")
Test1	1	4	1	valid
Test 2	3	3	3	invalid

The *space* program takes as input a file in the ADL format. We wrote a program to convert the file from the ADL format to the XML format. Then, we generated random XML files using Oxygen, which are converted back to the ADL format.

The *Apache Ant* program takes as input a XML file. The DTD for this XML file is available. However, we did not model all the tasks[1] of the *Apache Ant*. Thus tasks that are not modeled in our approach were removed from the DTD file. Table 4 shows the number of test cases generated for each subject program.

Table 4: Number of test cases

Testing method / **Subject program**	**2-way**	**3-way**
Ant	836	2121
Space	120	315

D. Metrics

Two metrics are used to measure the effectiveness of each approach. The first metric is statement coverage. We use *clover* [7] to collect code coverage information for *Apache Ant*, which is written in java, and *gcov* [18] for *Space*, which is written in C.

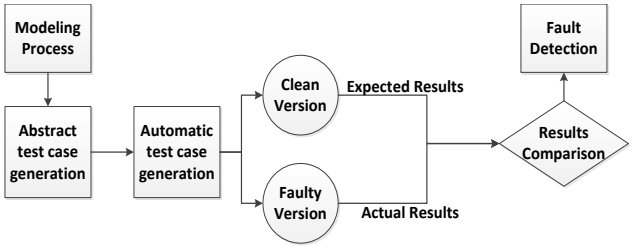


Fig.7: Fault detection procedure

[1] We only modeled common tasks of *Apache Ant* such as archive, compile, documentation, exestuation, file tasks and logging tasks

The second metric is the number of faults detected by each approach. Each of the two subject programs has multiple versions available in the SIR repository: one clean version and several faulty versions. Each faulty version contains a single fault (Fig.7). We count the number of faulty version that can be detected by each approach.

E. Results and discussion

For the pure-random approach, we first tried to generate random XML files using Oxygen solely based on the DTD file, i.e., without supplying any additional information. This approach only achieved 22% statement coverage on average. Since this approach is so ineffective, we do not consider it in the rest of our experiments.

To make the pure-random approach more meaningful, we provided additional information to the random XML files generation process. There are two types of additional information: (1) Information about the environment such as directory name, file name, class path, etc.; and (2) Constraints that may exist between different elements, e.g. uniqueness constraints, cross-reference constraints, etc.

For example the following is part of the original *Apache Ant* DTD file:

```
<!ELEMENT PROJECT (TARGET)+>
<!ATTLIST PROJECT
      Name   CDATA #IMPLIED
      Basedir CDATA #REQUIRED >
```

The PROJECT element has two attributes *Name* and *Basedir* listed in ATTLIST. The CDATA element indicates that the value is a character data. The REQUIRED or IMPLIED element indicates that the value is required or not.

The Name attribute represents the name of the project, which is an optional value. The Basedir attribute represents the base directory of the project. The Basedir attribute is required and cannot be a random string. This is because the *Apache Ant* program terminates if the base directory does not exist. Therefore, we modified the DTD file and fixed the value of Basedir to current directory.

The pure-random approach with such semantic information added to the schema achieved 53% statement coverage on average (Fig.8). While this is a significant improvement over the pure-random approach without any additional information, there is still a lot of room for improvement.

The modeled-random approach used the same model as our combinatorial testing approach, but instead of generating t-way abstract test cases using ACTS, it used MS Excel to generate the random abstract test cases.

The results in Fig.8 show that our approach achieved higher code coverage than the modeled-random approach, which further achieved higher code coverage than the pure-random approach. The results in Fig.8 also show that 3-way testing achieved higher code coverage than 2-way testing.

We also conducted an investigation to find out how many t-way combinations are covered with the test set generated by the modeled-random approach. The results show that with

the same number of test cases as t-way testing, modeled-random covers more than 90% of (t-1)-way combinations and 81% to 87% of t-way combinations. This explains in part why the modeled-random approach achieved code coverage competitive to our combinatorial testing approach.

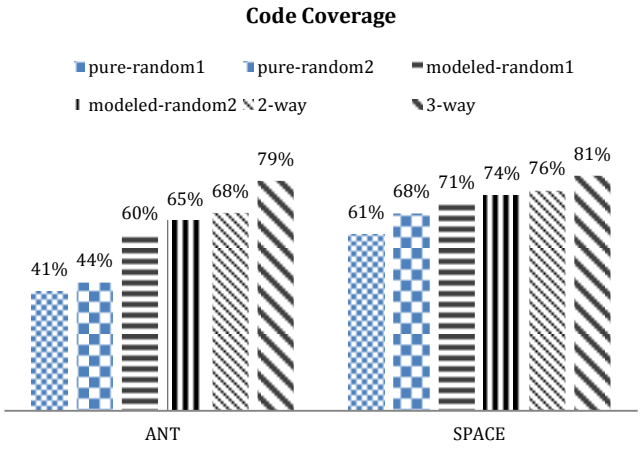


Fig.8: Code coverage results
The number of test cases for pure-random1 and modeled-random1 is the same as 2-way testing, and the number of test cases for pure-random2 and modeled-random2 is the same as 3-way testing.

After we executed the test cases, we inspected the faults to see how many versions we killed by looking at the source code. (We did not look at the source code during the modeling process.) Some of the faults are only triggered by invalid inputs and since we focused on interaction testing, we exclude those faults that can only be triggered by single invalid values. Therefore, we have 6 faulty versions for *Apache Ant* and 32 faulty versions for the *Space*.

Table 5: Fault detection results

Subject Programs	Ant		Space	
	killed	not killed	killed	not killed
pure-random1	1	5	12	20
pure-random2	1	5	15	17
modeled-random1	4	2	23	9
modeled-random2	4	2	26	6
2-way	4	2	28	4
3-way	5	1	30	2

Table 5 shows the fault detection results of the different approaches. These results are largely consistent with the code coverage results. That is, our approach detected more faults than modeled-random testing which further detected more faults than pure-random testing.

The modeled-random testing for *Space* detected a new fault with higher interaction strength. The notion of fault strength or degree of fault is introduced to show the number of parameters that are involved in causing the fault. The detail of this fault is explained below.

Table 6 shows a part of the fault detection table for *Space*. This table only shows faults that were not detected by at least one of our testing approaches. In other words, faults that were detected by all the three approaches are not shown. 3-way testing was able to detect 93.7% of the faults. The faulty versions 12 and 18 (v12 and v18) were only killed by 3-way testing. None of our tests was able to detect v27. Version v33 was only detected by modeled-random testing (with the same number of tests as 2-way testing).

To find out why some faults were not detected, we conducted an investigation to determine the strength of the faults mentioned above. Our investigation suggests that the strengths of fault for v12, v18, v27, and v33, are likely to be 4, 5, 7, and 5, respectively. This explains why they were not detected by some approaches. Note that a t-way test set also contains higher strength combinations. This is why v12 and v18 were detected by 3-way testing, even though they have a strength higher than 3. Similarly, the v33 was detected by modeled-random testing.

It is important to note that it can be difficult to determine the strength of fault for a large and/or complex program. In the following, we use v33 as an example to show how we determined the strength of a fault. The test case that killed the fault v33 includes 10 parameters. (Table 7)

In order to identify the suspicious parameters, we generated 20 more test cases by changing one parameter at a time and fixing the others. 8 out of 20 test cases were able to kill the version. By comparing the parameter values of these test cases, we were able to detect five suspicious parameters that could cause the fault.

To determine the strength of the fault, we generated 486 exhaustive test cases by fixing the value of the suspicious parameters. We randomly executed 10 out of 846, which they all failed. Therefore, we believe the strength of this fault is likely to be 5. [42]

We performed a similar investigation for v12 and v18 which both were killed only by 3-way testing.

In addition, we performed an investigation for v27. This version was not killed by any of our tests. The code coverage data showed that 14 out of 315 test cases executed the faulty statement. We traced the source code while executing the identified test cases. We applied the same method as described above.

Although a 3-way test set guarantees to kill the faulty version when the fault strength does not go over 3, but it is possible that a 3-way test set kills a version with fault strength greater than 3. Hence 3-way testing was able to kill v12 and v18.

Table 6: Part of the fault detection table for Space (killed=1, not killed=0)

Version #	**v7**	**v8**	**v12**	**v13**	**v16**	**v18**	**v20**	**v21**	**v22**	**v27**	**v29**	**v31**	**v33**	**v35**	**v36**	**v37**
pure-random1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
pure-random2	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1
modeled-random1	0	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1
modeled-random2	0	0	0	1	1	0	1	1	1	0	1	1	1	0	0	1
2-way	1	1	0	1	1	0	1	1	1	0	1	1	0	1	1	1
3-way	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1

Table 7: Test case parameters of v33

Test Factors	Test Values
grid	square
	triang
	rectang
	hex
geometry	rect
	circle
geop	>0
	<0
	=0
geoQ	>0
	<0
	=0
polarization	NA
	linepol
	circlpol
add	node
	block
	poly
	hex
THETA	NA
	>0
	<0
	=0
PHI	NA
	>0
	<0
	=0
PSI	NA
	>0
	<0
	=0
phase	NA
	uni
	secor
	rotate
	point
	pqpha

V. THREATS TO VALIDITY

The main threat to external validity is that the two subject programs used in our experiments may not be representatives of true practice. We plan to conduct more experiments on real-life programs in the future.

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. Furthermore, since our experiments use open source programs, the validity of our results would be in jeopardy if knowledge of the source code were used to identify these parameters and values in our experiments. To alleviate this potential threat, we only used the source code information to inspect the faults after our testing process is completed.

VI. CONCLUSION AND FUTURE WORK

In this paper we presented an input space modeling methodology for combinatorial testing. Input space modeling is problem zero of combinatorial testing, and it determines to a large extent the effectiveness of combinatorial testing. The key idea of our methodology is to consider the modeling process as two steps, input structure modeling and input parameter modeling. We mainly considered the graph structure, which is further divided into graphs without loop and graphs with loop. We also suggest some guidelines to perform unit and integration testing based on the graph structure. We believe that input structure modeling is essential to manage complex input spaces such as those represented by XML files.

We also reported two experiments of applying our methodology to two real-life programs. The results showed that combinatorial testing achieved higher code coverage and detected more faults than modeled-random testing. Both of these two approaches used the proposed methodology to model the input space and generated the same number of tests cases from the same model. In addition, the results show that both combinatorial testing and modeled-random testing are significantly more effective, in terms of code coverage and fault detection, than pure-random testing. This suggests that input space modeling is an essential step in the testing process.

We plan to conduct more studies for other real-life programs. The goal is to further validate the proposed methodology and develop a set of guidelines that can be used by 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 referenced in this paper or imply that the referenced product is necessarily the best.

VII. REFRENCES

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

[2]. 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.

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

[4]. 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.

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

[6]. 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.

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

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

[9]. C. Nie and H. Leung. A survey of combinatorial testing. ACM Computing Surveys (CSUR), 43:11:1–11:29, 2011

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

[11]. Burr, K. and Young, W.. 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, 1998.

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

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

[14]. Williams, A. W. And Probert, R. L.. 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, 1996.

[15]. Burroughs, K., Jain, A., and Erickson, R.. 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., 1994.

[16]. 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.

[17]. http://ant.apache.org/manual/

[18]. http://sourceforge.net/projects/gcov-eclipse/

[19]. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel, Empirical Software Engineering: An International Journal, Volume 10, No. 4, pages 405-435, 2005.

[20]. Borazjany, Mehra N., Yu Lei.,et al. "Combinatorial Testing of ACTS: A Case Study." In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 591-600. IEEE, 2012.

[21]. Richard Kuhn, Raghu Kacker, Yu Lei., Combinatorial and Random Testing Effectiveness for a Grid Computer Simulator presented at the Mod Sim World, Virginia, USA, 2009.

[22]. Mario Brčić and Damir Kalpić., Combinatorial testing in software projects Jubilee 35th International Convention Proceedings / Biljanović, Petar (ur.). - Rijeka : Croatian Society for Information and Communication Technology, Electronics and Microelectronics - MIPRO. 1832-1837, 2012.

[23]. P. J. Schroeder, P. Bolaki, and V. Gopu, Comparing the fault detection effectiveness of n-way and random test suites, International Symposium on Empirical Software Engineering, 2004. ISESE '04. Proceedings, pp. 49– 59, 2004.

[24]. Jones, James A., Mary Jean Harrold, and John Stasko. "Visualization of test information to assist fault localization." In Proceedings of the 24th international conference on Software engineering, pp. 467-477. ACM, 2002.

[25]. http://oxygenxml.com/xml_developer.html

[26]. C. Lott, A. Jain, S. Dalal, Modeling Requirements for Combinatorial Software Testing, SIGSOFT Softw. Eng. Notes,30:1-7, 2005

[27]. Itai Segall, Rachel Tzoref-Brill, Aviad Zlotnick, Common Patterns in Combinatorial Models , 1st International Workshop on Combinatorial Testing (in conjunction with ICST'12), 2012.

[28]. Itai Segall, Rachel Tzoref-Brill, Aviad Zlotnick, Simplified Modeling of Combinatorial Test Spaces, 1st International Workshop on Combinatorial Testing (in conjunction with ICST'12), 2012.

[29]. Elke Salecker, Sabine Glesner, Combinatorial Interaction Testing for Test Selection in Grammar-Based Testing, IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012.

[30]. D. Richard Kuhn, Raghu N. Kacker, Yu Lei , Practical Combinatorial Testing, National Institute of Standards and Technology, 2010.

[31]. Bryce, R. C., Lei, Y., Kuhn, D. R. & Kacker, R. Combinatorial Testing, Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization. IGI Global, 196-208., 2010

[32]. W. Wang, Y. Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. Kacker and R. Kuhn, "A combinatorial approach to detecting buffer overflow vulnerabilities", Proceedings of 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2011.

[33]. W. E. Wong, Y. Lei, Reachability Graph-Based Test Sequence Generation for Concurrent Programs, International Journal on Software Engineering and Knowledge Engineering, 18(6):803-822, Sept. 2008.

[34]. Y. Lei, R. Carver, R. Kacker, D. Kung, "A Combinatorial Strategy for Testing Concurrent Programs", Journal of Software Testing,Verification, and Reliability, 17(4):207-225, 2007.

[35]. Qu, Xiao, Myra B. Cohen, and Katherine M. Woolf. "Combinatorial interaction regression testing: A study of test case generation and prioritization." Software Maintenance, 2007. ICSM 2007. IEEE International Conference on. IEEE, 2007.

[36]. S. Dalal and C. L. Mallows, "Factor-Covering Designs for Testing Software," Technometrics, vol. 50, no. 3, pp. 234-243, 1998.

[37]. W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness," in Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering. Monterey, CA, pp. 230-238., 1994.

[38]. K. Burroughs, A. Jain, and R. L. Erickson, "Improved Quality of Protocol Testing Through Techniques of Experimental Design," in Proc. Supercomm./IEEE International Conference on Communications, pp. 745-752, 1994.

[39]. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 437-444, 1997.

[40]. D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, vol. 13, no. 5, pp. 83-88, 1996.

[41]. Yin, Huifang, Zemen Lebne-Dengel, and Yashwant K. Malaiya. "Automatic test generation using checkpoint encoding and antirandom testing." In Proc. The Eighth International Symposium On Software Reliability Engineering, pp. 84-95. IEEE, 1997.

[42]. Zhang, Zhiqiang, and Jian Zhang. "Characterizing failure-causing parameter interactions by adaptive testing." In Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 331-341. ACM, 2011.

[43]. J. Czerwonka. Pairwise Testing in Real World. In Proc. 24th Pacific Northwest Software Quality Conference (PNSQC'06), pages 419–430, 2006.