Test Algebra for Combinatorial Testing

Wei-Tek Tsai*, Charles J. Colbourn*†, Jie Luo†, Guanqiu Qi*, Qingyang Li*, Xiaoying Bai‡
*School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
†State Key Laboratory of Software Development Environment
School of Computer Science and Engineering,
Beihang University, Beijing, China
‡Department of Computer Science and Technology, INLIST
Tsinghua University, Beijing, China
{wtsai, colbourn}@asu.edu, luojie@nlsde.buaa.edu.cn
{guanqiuq, qingyan2b}@asu.edu, baixy@tsinghua.edu.cn

***Abstract*—This paper proposes a new algebraic system, Test Algebra (TA), for identifying faults in combinatorial testing for SaaS (Software-as-a-Service) applications. SaaS as a part of cloud computing is a new software delivery model, and mission-critical applications are composed, deployed, and executed in cloud platforms. Testing SaaS applications is a challenging task because new applications need to be tested when they are composed before they can be deployed for execution. Combinatorial testing algorithms can be used to identify faulty configurations and interactions from 2-way all the way to k-way where k is the number of components in the application. The TA defines rules to identify faulty configurations and interactions. Using the rules defined in the TA, a collection of configurations can be tested concurrently in different servers and in any order and the results obtained will be still same due to the algebraic constraints.**

***Index Terms*—Combinatorial testing, algebra, SaaS**

I. Introduction

Software-as-a-Service (SaaS) is a new software delivery model. SaaS often supports three features: customization, multi-tenancy architecture (MTA), and scalability. MTA means using one code base to develop multiple tenant applications, and each tenant application essentially is a customization of the base code [12]. A SaaS system often also supports scalability as it can supply additional computing resources when the workload is heavy. Tenants' applications are often customized by using components stored in the SaaS database [14], [1], [11] including GUI, workflow, service, and data components.

Once tenant applications are composed, they need to be tested, but a SaaS system can have millions of components, and hundreds of thousands of tenant applications. Testing tenant applications becomes a challenge as new tenant applications and components are added into the SaaS system continuously. New tenant applications are added on a daily basis while other tenant applications are running on the SaaS platform. As new tenant applications are composed, new components are added into the SaaS system. Each tenant application represents a customer for the SaaS system, and thus it needs to be tested.

Combinatorial testing is a popular testing technique to test an application with different configurations. It often assumes that each component within a configuration has been tested already, but interactions among components in the configuration may cause failures. Traditional combinatorial testing techniques often focus on test case generation to detect the presence of faults, but fault location is an active research area. Each configuration needs to be tested, as each configuration represents a tenant application. Traditional combinatorial testing such as *AETG* [2] can reveal the existence of faults by using few test cases to support t-way coverage for $t \geq 2$. But knowing the existence of a fault does not indicate which t-way interactions are faulty. When the problem size is small, an engineer can identify faults. However, when the problem is large, it can be a challenge to identify faults.

As the IT industries moves to Big Data and cloud computing where hundreds of thousand processors are available, potentially, a large number of processors with distributed databases can be used to perform large-scale combinatorial testing. Furthermore, they also provide concurrent and asynchronous computing mechanisms such as MapReduce, automated redundancy and recovery management, automated resource provisioning, and automated migration for scalability. These capabilities provide significant computing power that was not available before. One simple way of performing combinatorial testing in a cloud environment is:

1) Partition the testing tasks;
2) Allocate these testing tasks to different processors in the cloud platform for test execution;
3) Collect results done by these processors.

However, this is not efficient as while the number of computing and storage resources have increased significantly, the number of combinations to be considered is still too high. For example, a large SaaS system may have millions of components, and testing all of these combinations can still consume all the resources in a cloud platform. Two ways to improve this approach and both are based on learning from the previous test results:

- Devise a mechanism to merge test results from different processors so that testing results can be merged quickly, and detect any inconsistency in testing;
- Based on the existing testing results, eliminate any con-

figurations or interactions from future testing.

Due to the asynchronous and autonomous nature of cloud computing, test results may arrive asynchronously and autonomously, and this necessitates a new testing framework. This paper proposes a new algebraic system, Test Algebra (*TA*), to facilitate concurrent combinatorial testing. The key feature of *TA* is that the algebraic rules follow the combinatorial structure, and thus can track the test results obtained. The *TA* can then be used to determine whether a tenant application is faulty, and which interactions need to be tested. The *TA* is an algebraic system in which elements and operations are formally defined. Each element represents a unique component in the SaaS system, and a set of components represents a tenant application. Assuming each component has been tested by developers, testing a tenant application is equivalent to ensuring that there is no t-way interaction faults for $t \geq 2$ among the elements in a set.

The *TA* uses the principle that if a t-way interaction is faulty, every $(t+1)$-way interaction that contains the t-way interaction as a subset is necessarily faulty. The *TA* provides guidance for the testing process based on test results so far. Each new test result may indicate if additional tests are needed to test a specific configuration. The *TA* is an algebraic system, primarily intended to track the test results without knowing how these results were obtained. Specifically, it does not record the execution sequence of previously executed test cases. Because of this, it is possible to allocate different configurations to different processors for execution in parallel or in any order, and the test results are merged following the *TA* rules. The execution order and the merge order do not affect the merged results if the merging follows the *TA* operation rules.

This paper is structured as follows: Section II discusses the related work; Section III proposes *TA* and shows its details; and Section IV concludes this paper. Appendix provides proofs of TA associativity properties.

II. Related Work

SaaS testing is a new research topic [14], [4], [10]. Using policies and metadata, test cases can be generated to test SaaS applications. Testing can be embedded in the cloud platform where tenant applications are run [14]. Gao proposed a framework for testing cloud applications [4], and proposed a scalability measure for testing cloud application scalability. Another scalability measure was proposed by [10].

Testing all combinations of inputs and preconditions is not feasible, even with a simple product [6], [8]. The number of defects in a software product can be large, and defects occurring infrequently are difficult to find [15]. Combinatorial test design is used to identify a small number of tests needed to get the coverage of important combinations. Combinatorial test design methods enable one to build structure variation into test cases for having greater test coverage with fewer test cases.

Determining the presence of faults caused by a small number of interacting elements has been extensively studied in component-based software testing. When interactions are to be examined, testing involves a combination-based strategy [5]. When every interaction among t or fewer elements is to be tested, methods have been developed that provide pairwise or t-way coverage. Among the early methods, $AETG$ [2] popularized greedy one-test-at-a-time methods for constructing test suites. In the literature, the test suite is usually called a covering array, defined as follows. Suppose that there are k configurable elements, numbered from 1 to k. Suppose that for element c, there are v_c valid options. A t-way interaction is a selection of t of the k configurable elements, and a valid option for each. A test selects a valid option for every element, and it covers a t-way interaction if, when one restricts the attention to the t selected elements, each has the same option in the interaction as it does in the test.

A covering array of strength t is a collection of tests so that every t-way interaction is covered by at least one of the tests. Covering arrays reveal faults that arise from improper interaction of t or fewer elements [9]. There are numerous computational and mathematical approaches for construction of covering arrays with a number of tests as small as possible [3], [7].

If a t-way interaction causes a fault, then executing all tests of a covering array will reveal the presence of at least one faulty interaction. SaaS testing is interested in identifying those interactions that are faulty including their numbers and locations, as faulty configurations cannot be used in tenant applications. Furthermore, the number and location of faults keep on changing as new components can be added into the SaaS database continuously. By then executing each test, certain interactions are known not to be faulty, while others appear only in tests that reveal faults, and hence may be faulty. At this point, a classification tree analysis builds decision trees for characterizing possible sets of faults. This classification analysis is then used either to permit a system developer to focus on a small collection of possible faults, or to design additional tests to further restrict the set of possible faults. In [16], empirical results demonstrate the effectiveness of this strategy at limiting the possible faulty interactions to a manageable number. Assuming that interactions of more than t elements do not produce faults, a covering array can use few tests to certify that no fault arises from a t-way interaction.

The Adaptive Reasoning algorithm (*AR*) is a strategy to detect faults in SaaS [13]. The algorithm uses earlier test results to generate new test cases to detect faults in tenant applications. It uses three principles:

- **Principle 1:** When a tenant application (or configuration) fails the testing, there is at least one fault (but there may be more) in the tenant configuration.
- **Principle 2:** When a tenant application passes the testing, there is no fault in the tenant configuration resulting from a t-way interactions among components in the configuration.
- **Principle 3:** Whenever a configuration contains one or more faulty interactions, it is faulty.

III. Test Algebra

Let $\mathcal{C}$ be a finite set of *components*. A *configuration* is a subset $\mathcal{T} \subseteq \mathcal{C}$. One is concerned with determining the operational status of configurations. To do this, one can execute certain tests; every *test* is a configuration, but there may be restrictions on which configurations can be used as tests. If a certain test can be executed, its execution results in an outcome of *passed* (*operational*) or *failed* (*faulty*).

When a test execution yields result, all configurations that are subsets of the test are operational. However, when a test execution yields a faulty result, one only knows that at least one subset causes the fault, but it is unclear which of these subsets caused the failure. Among a set of configurations that may be responsible for faults, the objective is to determine, which cause faults and which do not. To do this, one must identify the set of candidates to be faulty. Because faults are expected to arise from an interaction among relatively few components, one considers t-*way interactions*. The t-way interactions are $\mathcal{I}_t = \{U \subseteq \mathcal{C} : |U| = t\}$. Hence the goal is to select tests, so that from the execution results of these tests, one can ascertain the status of all t-way interactions for some fixed small value of t.

Because interactions and configurations are represented as subsets, one can use set-theoretic operations such as union, and their associated algebraic properties such as commutativity, associativity, and self-absorption. The structure of subsets and supersets also plays a key role.

To permit this classification, one can use a *valuation function* V, so that for every subset S of components, $V(S)$ indicates the current knowledge about the operational status consistent with the components in S. The focus is on determining $V(S)$ whenever S is an interaction in $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_t$. These interactions can have one of five states.

- **Infeasible (**`X`**):** For certain interactions, it may happen that no feasible test is permitted to contain this interaction. For example, it may be infeasible to select two GUI components in one configuration such that one says the wall is GREEN but the other says RED.
- **Faulty (**`F`**):** If the interaction has been found to be faulty.
- **Operational (**`P`**):** Among the rest, if an interaction has appeared in a test whose execution gave an operational result, the interaction cannot be faulty.
- **Irrelevant (**`N`**):** For some feasible interactions, it may be the case that certain interactions are not expected to arise, so while it is possible to run a test containing the interaction, there is no requirement to do so.
- **Unknown (**`U`**):** If neither of these occurs then the status of the interaction is required but not currently known.

Any given stage of testing, an interaction has one of five possible status indicators. These five status indicators are ordered by $\texttt{X} \succ \texttt{F} \succ \texttt{P} \succ \texttt{N} \succ \texttt{U}$ under a relation $\succ$, and it has a natural interpretation to be explained in a moment.

A. Learning from Previous Test Results

The motivation for developing an algebra is to automate the deduction of the status of an interaction from the status of tests and other interactions, particularly in combining the status of two interactions. Specifically, one is often interested in determining $V(\mathcal{T}_1 \cup \mathcal{T}_2)$ from $V(\mathcal{T}_1)$ and $V(\mathcal{T}_2)$. To do this, a binary operation $\otimes$ on $\{\texttt{X}, \texttt{F}, \texttt{P}, \texttt{N}, \texttt{U}\}$ can be defined, with operation table as follows:

$\otimes$	X	F	P	N	U
X	X	X	X	X	X
F	X	F	F	F	F
P	X	F	U	N	U
N	X	F	N	N	N
U	X	F	U	N	U

Using this definition, one can verify that the binary operation $\otimes$ has the following properties of commutativity and associativity.

$V(\mathcal{T}_1) \otimes V(\mathcal{T}_2) = V(\mathcal{T}_2) \otimes V(\mathcal{T}_1)$,
$V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3)$.

Using this operation, one observes that $V(\mathcal{T}_1 \cup \mathcal{T}_2) \succeq V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)$. It follows that

1) Every superset of an infeasible interaction is infeasible.
2) Every superset of a failed interaction is failed or infeasible.
3) Every superset of an irrelevant interaction is irrelevant, failed, passed, or infeasible.

A set S is an `X`-*implicant* if $V(S) = \texttt{X}$ but whenever $S' \subset S$, $V(S') \prec \texttt{X}$. The `X`-*implicants* provide a compact representation for all interactions that are infeasible. Indeed for any interaction $\mathcal{T}$ that contains an `X`-*implicant*, $V(\mathcal{T}) = \texttt{X}$. Furthermore, a set S is an `F`-*implicant* if $V(S) = \texttt{F}$ but whenever $S' \subset S$, $V(S') \prec \texttt{F}$. For any interaction $\mathcal{T}$ that contains an `F`-*implicant*, $V(\mathcal{T}) \succeq \texttt{F}$. In the same way, a set S is an `N`-*implicant* if $V(S) = \texttt{N}$ but whenever $S' \subset S$, $V(S') = \texttt{U}$. For any interaction $\mathcal{T}$ that contains an `N`-*implicant*, $V(\mathcal{T}) \succeq \texttt{N}$. An analogous statement holds for passed interactions, but here the implication is for subsets. A set S is a `P`-*implicant* if $V(S) = \texttt{P}$ but whenever $S' \supset S$, $V(S') \succeq \texttt{F}$. For any interaction $\mathcal{T}$ that is contained in a `P`-*implicant*, $V(\mathcal{T}) = \texttt{P}$.

Implicants are defined with respect to the current knowledge about the status of interactions. When a t-way interaction is known to be infeasible, failed, or irrelevant, it must contain an `X`-, `F`-, or `N`-*implicant*. By repeatedly proceeding from t-way to $(t+1)$-way interactions, then, one avoids the need for any tests for $(t+1)$-way interactions that contain any infeasible, failed, or irrelevant t-way interaction. Hence testing typically proceeds by determining the status of the 1-way interactions, then proceeding to 2-way, 3-way, and so on. The operation $\otimes$ is useful in determining the implied status of $(t+1)$-way interactions from the computed results for t-way interactions, by examining unions of the t-way and smaller interactions and determining implications of the rule that $V(\mathcal{T}_1 \cup \mathcal{T}_2) \succeq V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)$. Moreover, when adding further interactions to consider, all interactions previously tested that passed are contained in a `P`-*implicant*, and every $(t+1)$ interaction contained in one of these interactions can be assigned status `P`.

For example, $V(a,b) = \texttt{P}$, $V(a,e) = \texttt{X}$, and other 2-way interactions exist as atomic interaction in *TA*. Based on the defined $\otimes$ operation, values of t-way interactions can be deduced from the atomic interactions and their contained interactions, such as $V(a,b,e) \succeq V(a,b) \otimes V(a,e) = \texttt{X}$, i.e. $V(a,b,e) = \texttt{X}$.

The 3-way interaction (a,b,c) can have inferred results from 2-way interactions (a,b), (a,c), (b,c). If any contained 2-way interaction has value F, the determining value of 3-way is F, without further testing needed. But if all values of contained 2-way interactions are P, (a,b,c) the interaction needs to be tested. In this case, U needs to be changed to non-U such as F or P, assuming the 3-way is not X or N.

B. Changing Test Result Status

When testing a configuration with n components, one should test individual components, 2-way interactions, 3-way interactions, all the way to n-way interactions. Since any combination of interactions is relevant in this case, the status of any interaction can be either X, F, P, or U. The status of a configuration is determined by the status of all interactions.

1) If an interaction has status X (F), the configuration has status X (F).
2) If all interactions have status P, the configuration has status P.
3) If some interactions still have status U, further tests are needed.

It is important to determine when an interaction with status U can be deduced to have status F or P instead. It can never obtain status X or N once having had status U.

To change U to P: An interaction is assigned status P if and only if it is a subset of a test that leads to proper operation.

To change U to F: Consider the candidate $\mathcal{T}$, one can conclude that $V(\mathcal{T}) = \texttt{F}$ if there is a test containing $\mathcal{T}$ that yields a failed result, but for every other candidate interaction $\mathcal{T}'$ that appears in this test, $V(\mathcal{T}') = \texttt{P}$. In other words, the only possible explanation for the failure is the failure of $\mathcal{T}$.

C. Matrix Representation

Suppose that each individual component passed the testing. Then the operation table starts from 2-way interactions, then enlarges to t-way interactions step by step. During the procedure, many test results can be deduced from the existing results following *TA* rules. For example, all possible configurations of (a, b, c, d, e, f) can be expressed in the form of matrix, or operation table. First, we show the operation table for 2-way interactions. The entries in the operation table are symmetric and those on the main diagonal are not necessary. So only half of the entries are shown.

As shown in Figure 1, 3-way interactions can be composed by using 2-way interactions and components. Thus, following the *TA* implication rules, the 3-way interactions operation table is composed based on the results of 2-way combinations. Here, (a, b, c, d, e, f) has more 3-way interactions than 2-way interactions. As seen in Figure 1, a 3-way interaction can be obtained through different combinations of 2-way interactions and components. For example, $\{a,b,c\} = \{a\} \cup \{b,c\} = \{b\} \cup \{a,c\} = \{c\} \cup \{a,b\} = \{a,b\} \cup \{a,c\} = \{a,b\} \cup \{b,c\} = \{a,c\} \cup \{b,c\}$. $V(a) \otimes V(b,c) = V(c) \otimes V(a,b) = V(a,b) \otimes V(b,c) = \texttt{P} \otimes \texttt{P} = \texttt{U}$. But $V(b) \otimes V(a,c) = V(a,b) \otimes V(a,c) = V(b,c) \otimes V(a,c) = \texttt{P} \otimes \texttt{F} = \texttt{F}$. As *TA* defines the order of the five status indicators, the result should be the value with highest order. So $V(a,b,c) = \texttt{F}$.

$\otimes$	a	b	c	d	e	f
a		P	F	N	X	U
b			P	X	N	F
c				F	P	P
d					F	X
e						U
f						

D. Merging Concurrent Testing Results

One way to achieve efficient testing is to allocate (overlapping or non-overlapping) tenant applications into different clusters, and each cluster is sent to a different set of servers for execution. Once each cluster completes its execution, the test results can be merged. The testing results of a specific interaction $\mathcal{T}$ in different servers should satisfy the following constraints.

- If $V(\mathcal{T}) = \texttt{U}$ in one cluster, then in other clusters, the same $V(\mathcal{T})$ can be either F, P, N, or U.
- If $V(\mathcal{T}) = \texttt{N}$ in one cluster, then in other clusters, the same $V(\mathcal{T})$ can be either F, P, N, or U.
- If $V(\mathcal{T}) = \texttt{P}$ in one cluster, then the same $V(\mathcal{T})$ can be either P, N, or U in all clusters;
- If $V(\mathcal{T}) = \texttt{F}$ in one cluster, then in other clusters, the same $V(\mathcal{T})$ can be F, N, or U.
- If $V(\mathcal{T}) = \texttt{X}$ in one cluster, then in other clusters, the same $V(\mathcal{T})$ can be X only.

If these constraints are satisfied, then the testing results can be merged. Otherwise, there must be an error in the testing results. To represent this situation, a new status indicator, error (E), is introduced and $\texttt{E} \succ \texttt{X}$. We define a binary operation $\oplus$ on $\{\texttt{E},\texttt{X},\texttt{F},\texttt{P},\texttt{N},\texttt{U}\}$, with operation table as follows:

$\oplus$	E	X	F	P	N	U
E	E	E	E	E	E	E
X	E	X	E	E	E	E
F	E	E	F	E	F	F
P	E	E	E	P	P	P
N	E	E	F	P	N	U
U	E	E	F	P	U	U

$\oplus$ also has the properties of commutativity and associativity. See Appendix for proof of associativity.

Using this operation, merging two testing results from two different servers can be defined as $V_{\text{merged}}(\mathcal{T}) = V_{\text{cluster1}}(\mathcal{T}) \oplus V_{\text{cluster2}}(\mathcal{T})$. The merge can be performed in any order due to the commutativity and associativity of $\oplus$, and if the constraints of merge are satisfied and $V(\mathcal{T})$ = X, F, or P, the results cannot be changed by any further testing or merging of test results unless there are some errors in testing. If $V(\mathcal{T})$ = E, the testing

$\cup$	a	b	c	$\cdots$	f	(a,b)	(a,c)	$\cdots$	(b,c)	$\cdots$	(e,f)
a	(a)	(a,b)	(a,c)	$\cdots$	(a,f)	(a,b)	(a,c)	$\cdots$	(a,b,c)	$\cdots$	(a,e,f)
b		(b)	(b,c)	$\cdots$	(b,f)	(a,b)	(a,b,c)	$\cdots$	(b,c)	$\cdots$	(b,e,f)
c			(c)	$\cdots$	(c,f)	(a,b,c)	(a,c)	$\cdots$	(b,c)	$\cdots$	(c,e,f)
$\vdots$				$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
f					(f)	(a,b,f)	(a,c,f)	$\cdots$	(b,c,f)	$\cdots$	(e,f)
(a,b)						(a,b)	(a,b,c)	$\cdots$	(a,b,c)	$\cdots$	(a,b,e,f)
(a,c)							(a,c)	$\cdots$	(a,b,c)	$\cdots$	(a,c,e,f)
$\vdots$								$\ddots$	$\vdots$	$\vdots$	$\vdots$
(b,c)									(b,c)	$\cdots$	(b,c,e,f)
$\vdots$										$\ddots$	$\vdots$
(e,f)											(e,f)

$\otimes$	a	b	c	$\cdots$	f	(a,b)	(a,c)	$\cdots$	(b,c)	$\cdots$	(e,f)
a		P	F	$\cdots$	U	U	F	$\cdots$	U	$\cdots$	U
b			P	$\cdots$	F	U	F	$\cdots$	U	$\cdots$	U
c				$\cdots$	P	U	F	$\cdots$	U	$\cdots$	U
$\vdots$				$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
f						U	F	$\cdots$	U	$\cdots$	U
(a,b)							F	$\cdots$	U	$\cdots$	U
(a,c)								$\cdots$	F	$\cdots$	F
$\vdots$								$\ddots$	$\vdots$	$\vdots$	$\vdots$
(b,c)										$\cdots$	U
$\vdots$										$\ddots$	$\vdots$
(e,f)											

Fig. 1. 3-way interaction operation table

environment must be corrected and tests executed again after fixing the error(s) in testing. For example, $V_{\text{cluster1}}(a,c,e) = \texttt{X}$ and $V_{\text{cluster2}}(a,c,e) = \texttt{F}$, so $V_{\text{merged}}(a,c,e) = \texttt{X} \oplus \texttt{F} = \texttt{E}$. It means that there is something wrong with the tests of interaction (a,c,e), and the problem must be fixed before doing further testing.

Following the $\oplus$ associative rule, one can derive the following.

$$
\begin{aligned}
& V_1(\mathcal{T}) \oplus V_2(\mathcal{T}) \oplus V_3(\mathcal{T}) \\
&= (V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) \\
&= V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) \\
&= V_1(\mathcal{T}) \oplus V_2(\mathcal{T}) \oplus V_3(\mathcal{T}) \oplus V_3(\mathcal{T}) \\
&= (V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) \\
&= ((V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) \\
&= (V_3(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus (V_3(\mathcal{T}) \oplus V_1(\mathcal{T}))
\end{aligned}
$$

Thus the $\oplus$ rule allows one to partition the configurations into different sets for different servers to run testing, and these sets do not need to be non-overlapping. In conventional cloud computing operations such as MapReduce, data should not overlap, otherwise incorrect data may be produced. For example, counting the items in a set can be performed by MapReduce, but data allocated to different servers cannot overlap, otherwise items may be counted more than once. In *TA*, this is not a concern due to the nature of the *TA* operations $\oplus$. Once the results are available from each server, the testing results can be merged either incrementally, in parallel, or in any order. Furthermore, test results can be merged repeatedly without changing the final results.

Using this approach, one can test say 10 batches of 100 tenant applications each, for a total of $1,000$ tenant applications. Once this is done, another batch of $1,000$ tenant applications can be tested with each 100 tenant application allocated to a server for execution. In this way, after running 100 batches, $100,000$ tenant applications can be evaluated completely.

The following example illustrates the testing process of fifteen configurations. According to the definition of configuration testing, all feasible interactions should be tested. For simplicity, assume that only interaction (c,d,f) is faulty, and only interaction (c,d,e) is infeasible, and all other interactions pass the testing. If one assigns configurations 1, 3, 5, 7, 9, 11, 13, 15 into $Server_1$, configurations 2, 4, 6, 8, 10, 12, 14 into $Server_2$, and 4-11 configurations into $Server_3$.

If $Server_1$ and $Server_3$ do their own testing first, $Server_2$ can reuse test results of interactions from them to eliminate interactions that need to be tested. For example, when testing 2-way interactions of configuration (b,c,d,f) in $Server_2$, it can reuse the test results of (b,c), (b,d) of configuration (b,c,d,e) from $Server_3$, (b,f) of configuration (a,b,c,f)

from $Server_1$. They are all passed, and it can reuse the test results of (b, c, d) of configuration (a, b, c, d) from $Server_1$, (b, c, f) of configuration (a, b, c, f) from $Server_1$, (b, d, f) of configuration (a, b, d, f) from $Server_1$, and (c, d, f) of configuration (a, c, d, f) from $Server_3$. Because (c, d, f) is faulty, it can deduce that 4-way interaction (b, c, d, f) is also faulty. For the sets of configuration that are overlapping, their returned test results from different servers are the same. The merged results of these results also stay the same.

Not only interactions, sets of configurations, CS_1, CS_2, $\ldots, CS_K$ can be allocated to different processors (or clusters) for testing, and the test results can then be merged. The sets can be non-overlapping or overlapping, and the merge process can be arbitrary. For example, say the result of CS_i is RCS_i, the merge process can be $(\cdots((((RCS_1 + RCS_2) + RCS_3) + RCS_4) + \cdots + RCS_K)$, or $(\cdots((((RCS_K + RCS_{k-1}) + RCS_{k-2}) + \cdots + RCS_1)$, or any other sequence that includes all RCS_i, for $i = 1$ to K. This is true because RCS is simply a set of $V(\mathcal{T}_j)$ for any intercation $\mathcal{T}_j$ in the configuration CS_i.

	$Server_1$	$Server_2$	$Server_3$	$Merged\ Results$
(a,b,c,d)	P			P
(a,b,c,e)		P		P
(a,b,c,f)	P			P
(a,b,d,e)		P	P	P
(a,b,d,f)	P		P	P
(a,b,e,f)		P	P	P
(a,c,d,e)	X		X	X
(a,c,d,f)		F	F	F
(a,c,e,f)	P		P	P
(a,d,e,f)		P	P	P
(b,c,d,e)	X		X	X
(b,c,d,f)		F		F
(b,c,e,f)	P			P
(b,d,e,f)		P		P
(c,d,e,f)	X			X

E. Modified Testing Process

Perform 2-way interaction testing first. Before going on to 3-way interaction testing, use the results of 2-way testing to eliminate cases. The testing process stops when finishing testing all t-way interactions. The analysis of t-way interactions is based on the PTRs of all $(t - i)$-way interactions for $1 \leq i < t$. The superset of infeasible, irrelevant, and faulty test cases do not need to be tested. The test results of the superset can be obtained by *TA* operations and must be infeasible, irrelevant, or faulty. But the superset of test cases with unknown indicator must be tested. In this way, a large repeating testing workload can be reduced.

For n components, all t-way interactions for $t \geq 2$ are composed by 2-way, 3-way, ..., t-way interactions. In n components combinatorial testing, the number of 2-way interactions is equal to $\binom{n}{2}$. In general, the number of t-way interactions is equal to $\binom{n}{t}$. More interactions are treated when $\binom{n}{t} > \binom{n}{t-1}$, which happens when $t \leq \frac{n}{2}$. The total number of interactions examined is $\sum_{i=2}^{t} \binom{n}{i}$.

IV. Conclusion

This paper proposes *TA* to address SaaS combinatorial testing. The *TA* provides a foundation for concurrent combinatorial testing. The *TA* has two operations and test results can have five states with a priority. By using the *TA* operations, many combinatorial tests can be eliminated as the *TA* identifies those interactions that need not be tested. Also the *TA* defines operation rules to merge test results done by different processors, so that combinatorial tests can be done in a concurrent manner. The *TA* rules ensure that either merged results are consistent or a testing error has been detected so that retest is needed. In this way, large-scale combinatorial testing can be carried out in a cloud platform with a large number of processors to perform test execution in parallel to identify faulty interactions.

Acknowledgment

This project is sponsored by U.S. National Science Foundation project DUE 0942453 and National Science Foundation China (No.61073003), National Basic Research Program of China (No.2011CB302505), and the State Key Laboratory of Software Development Environment (No. SKLSDE-2012ZX-18), and Fujitsu Laboratory.

References

[1] X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao. Cloud Testing Tools. In *Proceedings of IEEE 6th International Symposium on Service Oriented System Engineering (SOSE)*, pages 1–12, Irvine, CA, USA, 2011.

[2] D. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *Journal of IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[3] C. J. Colbourn. Covering arrays and hash families. In D. Crnkovic and V. D. Tonchev, editors, *Information Security, Coding Theory and Related Combinatorics*, volume 29 of *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press, 2011.

[4] J. Gao, X. Bai, and W.-T. Tsai. SaaS Performance and Scalability Evaluation in Cloud. In *Proceedings of The 6th IEEE International Symposium on Service Oriented System Engineering*, SOSE '11, 2011.

[5] M. Grindal, J. Offutt, and S. F. Andler. Combination Testing Strategies: A Survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[6] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software, 2nd Edition*. Wiley, New York, NY, USA, 1999.

[7] V. V. Kuliamin and A. A. Petukhov. A Survey of Methods for Constructing Covering Arrays. *Journal of Program Computer Software*, 37(3):121–146, may 2011.

[8] T. Muller and D. Friedenberg. Certified Tester Foundation Level Syllabus. *Journal of International Software Testing Qualifications Board*.

[9] A. A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *IEEE Transactions on Software Engineering*, 33(8):510–525, 2007.

[10] W.-T. Tsai, Y. Huang, X. Bai, and J. Gao. Scalable Architecture for SaaS. In *Proceedings of 15th IEEE International Symposium on Object Component Service-oriented Real-time Distributed Computing*, ISORC '12, Apr. 2012.

[11] W.-T. Tsai, Y. Huang, and Q. Shao. Testing the Scalability of SaaS Applications. In *Proceedings of IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4, Irvine, CA, USA, 2011.

[12] W.-T. Tsai, Y. Huang, Q. Shao, and X. Bai. Data Partitioning and Redundancy Management for Robust Multi-Tenancy SaaS. *International Journal of Software and Informatics (IJSI)*, 4(3):437–471, 2010.

[13] W.-T. Tsai, Q. Li, C. J. Colbourn, and X. Bai. Adaptive Fault Detection for Testing Tenant Applications in Multi-Tenancy SaaS Systems. In *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*, March 2013.

[14] W.-T. Tsai, Q. Shao, and W. Li. OIC: Ontology-based Intelligent Customization Framework for SaaS. In *Proceedings of International Conference on Service Oriented Computing and Applications(SOCA'10)*, Perth, Australia, Dec. 2010.

[15] Wikipedia. Software Testing, 2013.

[16] C. Yilmaz, M. B. Cohen, and A. Porter. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 45–54, New York, NY, USA, 2004. ACM.

APPENDIX

The associativity of binary operation $\otimes$.

$$V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3).$$

Proof: We will prove this property in the following cases.

(1) At least one of $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ is X. Without loss of generality, suppose that $V(\mathcal{T}_1) = \mathtt{X}$, then according to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \mathtt{X} \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \mathtt{X}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = (\mathtt{X} \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \mathtt{X} \otimes V(\mathcal{T}_3) = \mathtt{X}$. Thus, in this case, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3)$.

(2) $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ are not X and at least one of $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ is F. Without loss of generality, suppose that $V(\mathcal{T}_1) = \mathtt{F}$, then according to the operation table of $\otimes$, the value of $V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)$ can only be F, N or U. So $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \mathtt{F} \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \mathtt{F}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = (\mathtt{F} \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \mathtt{F} \otimes V(\mathcal{T}_3) = \mathtt{F}$. Thus, in this case, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3)$.

(3) $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ are not X or F and at least one of $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ is N. Without loss of generality, suppose that $V(\mathcal{T}_1) = \mathtt{N}$, then according to the operation table of $\otimes$, the value of $V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)$ can only be N or U. So $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \mathtt{N} \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \mathtt{N}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = (\mathtt{N} \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \mathtt{N} \otimes V(\mathcal{T}_3) = \mathtt{N}$. Thus, in this case, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3)$.

(4) $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ are not X, F or N. In this case, $V(\mathcal{T}_1)$, $V(\mathcal{T}_2)$, and $V(\mathcal{T}_3)$ can only be P or U. According to the operation table of $\otimes$, the value of $V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)$ and $V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)$ are U. So $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = V(\mathcal{T}_1) \otimes \mathtt{U} = \mathtt{U}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \mathtt{U} \otimes V(\mathcal{T}_3) = \mathtt{U}$. Thus, in this case, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3)$. ■

The associativity of binary operation $\oplus$.

$$V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = (V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}).$$

Proof: We will prove this property in the following cases.

(1) One of $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ is E. Without loss of generality, suppose that $V_1(\mathcal{T}) = \mathtt{E}$, then according to the operation table of $\oplus$, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{E} \otimes (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{E}$, $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = (\mathtt{E} \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \mathtt{E} \oplus V_3(\mathcal{T}) = \mathtt{E}$. Thus, in this case, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = (V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T})$.

(2) $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ are not E, and there is a pair of $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ does not satisfy the constrains. Without loss of generality, suppose that $V_1(\mathcal{T})$ and $V_2(\mathcal{T})$ does not satisfy the constrains, then according to the operation table of $\oplus$, $V_1(\mathcal{T}) \oplus V_2(\mathcal{T}) = \mathtt{E}$. So $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \mathtt{E} \oplus V_3(\mathcal{T}) = \mathtt{E}$. Since $V_1(\mathcal{T})$ and $V_2(\mathcal{T})$ does not satisfy the constrains, there can be two cases: (a) one of them is X and the other is not, or (b) one of them is P and the other is F.

(a) If $V_1(\mathcal{T}) = \mathtt{X}$, then $V_2(\mathcal{T}) \oplus V_3(\mathcal{T})$ cannot be X because $V_2(\mathcal{T})$ cannot be X. Thus, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{E}$. If $V_2(\mathcal{T}) = \mathtt{X}$, then $V_2(\mathcal{T}) \oplus V_3(\mathcal{T}) \neq \mathtt{X}$ can only be E or X. Since $V_1(\mathcal{T})$ cannot be X, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{E}$.

(b) If $V_1(\mathcal{T}) = \mathtt{P}$ and $V_2(\mathcal{T}) = \mathtt{F}$, then $V_2(\mathcal{T}) \oplus V_3(\mathcal{T})$ can only be E or F. Thus, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{E}$. If $V_1(\mathcal{T}) = \mathtt{F}$ and $V_2(\mathcal{T}) = \mathtt{P}$, then $V_2(\mathcal{T}) \oplus V_3(\mathcal{T})$ can only be E or P. Thus, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{E}$.

Thus, in this case, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = (V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T})$.

(3) $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ are not E, and $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ satisfy the constrains.

(a) One of $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ is X. Without loss of generality, suppose that $V_1(\mathcal{T}) = \mathtt{X}$, then $V_2(\mathcal{T}) = V_3(\mathcal{T}) = \mathtt{X}$. So $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{X} \oplus (\mathtt{X} \oplus \mathtt{X}) = \mathtt{X} \oplus \mathtt{X} = \mathtt{X}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = (\mathtt{X} \oplus \mathtt{X}) \oplus \mathtt{X} = \mathtt{X} \oplus \mathtt{X} = \mathtt{X}$.

(b) $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ are not X, and one of $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ is F. Without loss of generality, suppose that $V_1(\mathcal{T}) = \mathtt{F}$, then $V_2(\mathcal{T})$ and $V_3(\mathcal{T})$ can only be F, N, or U. According to operation table of $\oplus$, $V_2(\mathcal{T}) \oplus V_3(\mathcal{T})$ can only be F, N, or U, and $V_1(\mathcal{T}) \oplus V_2(\mathcal{T})$ can only be F. So $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{F} \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{F}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \mathtt{F} \oplus V_3(\mathcal{T}) = \mathtt{F}$.

(c) $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ are not X or F, and one of $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ is P. Without loss of generality, suppose that $V_1(\mathcal{T}) = \mathtt{P}$, then $V_2(\mathcal{T})$ and $V_3(\mathcal{T})$ can only be P, N, or U. According to operation table of $\oplus$, $V_2(\mathcal{T}) \oplus V_3(\mathcal{T})$ can only be P, N, or U, and $V_1(\mathcal{T}) \oplus V_2(\mathcal{T})$ can only be F. So $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{P} \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{P}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \mathtt{P} \oplus V_3(\mathcal{T}) = \mathtt{P}$.

(d) $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ are not X, F or P, and one of $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ is U. Without loss of generality, suppose that $V_1(\mathcal{T}) = \mathtt{U}$, then $V_2(\mathcal{T})$ and $V_3(\mathcal{T})$ can only be N, or U. According to operation table of $\oplus$, $V_2(\mathcal{T}) \oplus V_3(\mathcal{T})$ can only be N, or U, and $V_1(\mathcal{T}) \oplus V_2(\mathcal{T})$ can only be U. So $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{U} \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{U}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \mathtt{U} \oplus V_3(\mathcal{T}) = \mathtt{U}$.

(e) $V_1(\mathcal{T})$, $V_2(\mathcal{T})$, and $V_3(\mathcal{T})$ are N. $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \mathtt{N} \oplus (\mathtt{N} \oplus \mathtt{N}) = \mathtt{N} \oplus \mathtt{N} = \mathtt{N}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = (\mathtt{N} \oplus \mathtt{N}) \oplus \mathtt{N} = \mathtt{N} \oplus \mathtt{N} = \mathtt{N}$.

Thus, in this case, $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = (V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T})$. ■