Test Algebra for Concurrent Combinatorial Testing

by

Guanqiu Qi

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

A new algebraic system, Test Algebra (TA), is proposed for identifying faults in combinatorial testing for SaaS (Software-as-a-Service) applications. In the context of cloud computing, SaaS is a new software delivery model, in which mission-critical applications are composed, deployed, and executed on cloud platforms. Testing SaaS applications is challenging because new applications need to be tested once they are composed, and prior to their deployment. A composition of components providing services yields a configuration providing a SaaS application. While individual components in the configuration may have been thoroughly tested, faults still arise due to interactions among the components composed, making the configuration faulty. When there are k components, combinatorial testing algorithms can be used to identify faulty interactions for t or fewer components, for some threshold $2 \leq t \leq k$ on the size of interactions considered. In general these methods do not identify specific faults, but rather indicate the presence or absence of some fault. To identify specific faults, an adaptive testing regime repeatedly constructs and tests configurations in order to determine, for each interaction of interest, whether it is faulty or not. In order to perform such testing in a loosely coupled distributed environment such as the cloud, it is imperative that testing results can be combined from many different servers. The TA defines rules to permit results to be combined, and to identify the faulty interactions. Using the TA, configurations can be tested concurrently on different servers and in any order. The results, using the TA, remain the same.

Dedicated to my grandparents, Amy, and Lin

ACKNOWLEDGEMENTS

I wish to express my great thanks to all the people who gave me tremendous support and help during my Ph.D. study.

First and foremost, I am heartily thankful to my advisor, Professor Wei-Tek Tsai. This dissertation would not have been possible without his guidance. From him, I learned how to find high-impact problems, conduct rigorous research, and make effective presentations. Dr.Tsai's brilliance makes my research much more enjoyable. His persistence and dedication encourage me to go through many obstacles. For me, Dr.Tsai is not just an advisor in research, but also whole life advisor. He gives me many advises in multiple aspects, especially on how to move beyond my comfort zone to reach my full potential. I would benefit from all these for my whole career.

I also thank my other dissertation committee members, Dr.Hasan Davulcu, Dr. Hessam Sarjoughian and Dr.Hongyu Yu, for their constructive suggestions for my dissertation and invaluable help for my career. Their suggestions make my dissertation more complete and accurate. Their recommendations open up many opportunities for my career.

Then, I would like to express my deep appreciation and gratitude to those collaborators for helping me finish my dissertation. Professor Charles Colbourn of Arizona State University formulated Test Algebra (TA). Professor Jie Luo of Beihang University provided TA proofs including various laws in TA. Professor Wenjun Wu and Chao Jin of Beihang University provided TA MapReduce simulation.

Then, many thanks to Dr.Yinong Chen for his continuous helps. He not only provided me teaching assistant position during my whole Ph.D. study, but also gave me precious suggestions on my research.

Furthermore, I am proud to be a member of ASU-SRLab. I benefit a lot from the discussions with all of the SRLabers, Dr. Qihong Shao, Wu Li, Xin Sun, Peide

Zhong, Le Xu, Yu Huang, and Jay Elston. Especially, thanks to Dr. Qihong Shao for her guidance and help in the beginning of my research. Their consistent supports in the past six years and a half help me handle all difficulties in my research and life.

Finally, I would like thank Amy for her support and encouragement. Without her consistent financial support, I could not finish my Ph.D. study. Also I would like to thank my grandparents and Lin for their deepest love, faith, understanding, and confidence in me. Without their consistent mentally supports, I could not finish my PhD study. This dissertation is dedicated to all of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Table Page

Chapter 1

INTRODUCTION

Cloud computing plays an important role today, as a new computing infrastructure to enable rapid delivery of computing resources as a utility in a dynamic, scalable, and visualized manner. Software-as-a-Service (SaaS), as a part of cloud computing among Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS), is a new software delivery model designed for Internet-based services. One single code is designed to run for different tenants. SaaS provides frequent upgrades to minimize customer disruption and enhance satisfaction. For maintenance, fixing one problem for one tenant also fixes it for all other tenants.

SaaS supports *customization*: Tenant applications are formed by composing components in the SaaS database Tsai *et al.* (2010b); Bai *et al.* (2011); Tsai *et al.* (2011) such as GUI, workflow, service, and data components. SaaS supports *multi-tenancy architecture (MTA)*: One code base is used to develop multiple tenant applications, so that each tenant application is a customization of the base code Tsai *et al.* (2010a). SaaS often also supports *scalability*, as it can supply additional computing resources when the workload is heavy.

Once tenant applications are composed, they need to be tested. However, a SaaS system can have millions of components, and hundreds of thousands of tenant applications. New tenant applications are added continuously, while other tenant applications are running on the SaaS platform. New tenant applications can cause new components to be added to the SaaS system.

Combinatorial testing Bryce *et al.* (2010) is a popular testing technique to test a component-based application. It tests interactions among components in the configuration, assuming that each component has been tested individually. A *t-way interaction* is one that involves t components, and *t-way coverage* in a test suite means that every t-way interaction appears in at least one test configuration. Traditional combinatorial testing techniques focus on tests 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 methods, such as *AETG* Cohen *et al.* (1997), 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 by knowing which test configurations contain a fault. However, when the problem is large, it difficult or even impossible to identify faults if the test suite only ensures t-way coverage.

The movement to Big Data and cloud computing can make hundreds of thousands of processors available. Potentially a large number of processors with distributed databases can be used to perform large-scale combinatorial testing. Indeed, these provide significant computing power that was not available before; for example, they support concurrent and asynchronous computing mechanisms such as MapReduce, automated redundancy and recovery management, automated resource provisioning, and automated migration for scalability. One simple way to perform 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 from the processors.

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

- Devise a mechanism to merge test results quickly, and detect any inconsistency in testing;
- Eliminate as many configurations as possible from future testing using existing testing results.

With cloud computing, test results may arrive asynchronously and autonomously. This necessitates a new testing framework. This paper proposes a new algebraic system, TA Tsai *et al.* (2013a), 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.

Chapter 2

RELATED WORK

2.1 Software Testing

Software testing is an essential activity in software development to ensure the correctness of program, or software quality Zhang *et al.* (2014). In general, testing is often an after-thought for a new technology, and it was not considered beforehand. Software testing uses different test cases to detect potential software bugs that cannot be identified during software development. Many testing methods have been proposed and used to increase the quality and reliability of software and systems Zhang *et al.* (2014); Mathur (2013). For example, black-box testing tests the functionality of an application without knowing its internal structures or workings Wikipedia (2014c) and white-box testing tests internal structures or workings of an application Wikipedia (2014g). Conventional software testing already faces significant complexity issues as number of data, paths, combinations, and permutations that are already large (exponential).

One main challenge of software testing is to represent the variability in an expressive and practical way. Domain-specific languages, feature diagrams, and other modeling techniques are used to express variability Sinnema and Deelstra (2007).

Another challenge is to generate test cases automatically using a description of the variability to reveal faults effectively. Testing all combinations of inputs and/or configurations is infeasible in general Kaner *et al.* (1999); Muller and Friedenberg (2007). The number of defects in a software product can be large, and defects occurring infrequently are difficult to find. Testing regimes balance the needs to generate tests quickly, to employ as few tests as possible, and to represent as many of the potential faults in tests as possible.

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 Grindal *et al.* (2005a). *Random testing* (see Arcuri and Briand (2012), for example) selects each test configuration (i.e., one choice for each component) randomly without reference to earlier selections. *Adaptive random testing* (*ART*) algorithms generate test suites use restricted random testing Huang *et al.* (2012) generate tests that are as "different" as possible from one another. Adaptive distance-based testing typically uses Hamming distance and uncovered combinations distance to generate combinatorial testing test suites. Parameters are ordered at random during the process of generating the next test case. Each parameter is assigned to a maximal value of the distance against the previously generated test cases Bryce *et al.* (2011).

2.2 Cloud Testing

Cloud computing plays an important role today. Many traditional softwares are hosted in cloud. The traditional software design has been changed, according to the new features of cloud.

- **Multi-tenancy architecture:** The software is designed to support multiple tenants to process their requirements at the same time. Each tenant shares the data, configuration, user management, and so on. Significant trade-offs exist between customization capability, security, and performance;

- **Sub-tenancy architecture:** It is another significant levels of complexity as tenant applications need to act as the SaaS infrastructure. Tenant application allows its own sub-tenant to develop applications. New issues includes sharing and security control, such as information flow;

- **Adaptive architecture and design:** Self-describing, self-adaptive, and tenant-aware units that can be migrated to any processors, also extend the design all the way to storage and network.

Cloud also introduces new testing issues. Not only new designs of cloud software need to be tested, but also testing tenant applications needs to involve SaaS infrastructure and as SaaS/PaaS often provides automated provisioning, scheduling, and built-in fault-tolerant computing including migration. Test engines need to monitor all changes in tenant application, such as increased/decreased resource, process relocation, and automated recovery. And additional resources may be needed to perform similar relocation to ensure testing completeness. Even running the same experiments in the same infrastructure may produce different performance and behaviors.

SaaS testing is a new research topic Tsai *et al.* (2010b); Gao *et al.* (2011b); Tsai *et al.* (2012). It is concerned with identifying those interactions that are faulty including their numbers and locations. Furthermore, the number and location of faults may change as new components are added to the SaaS database. Using policies and metadata, test cases can be generated to test SaaS applications. Testing can be embedded in the cloud platform in which tenant applications are run Tsai *et al.* (2010b). Gao proposed a framework for testing cloud applications Gao *et al.* (2011b), and proposed a measure for testing scalability. Another scalability measure was proposed in Tsai *et al.* (2012).

2.3 Combinatorial Designs

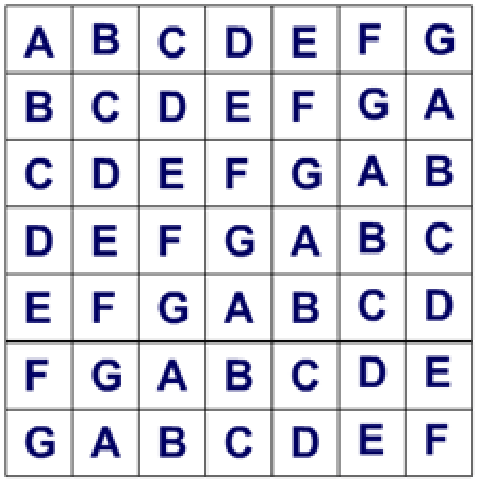
The concepts of combinatorial objects are not new to testing. The use of orthogonal arrays in statistically designed experiments are discussed Hedayat *et al.* (1999). Then the ideas are extended to different areas, including software testing. The combinatorial test suites are represented abstractly in mathematical and algorithmic way. A small number of test suites that covers many combinations of parameters is generated for the System Under Test (SUT). The following combinatorial designs are used.

2.3.1 Latin Square

A Latin square is an n*n array filled with n different symbols, each occurring exactly once in each row and exactly once in each column Wikipedia (2014e). One classic computable formula for the number of L(n) of n*n array is $\prod_{k=1}^{n}(k!)^{\frac{n}{k}} \geq L(n) \geq \frac{(n!)^{2n}}{n^{n^2}}$ van Lint and Wilson (1992). Figure 2.1 shows the 7*7 Latin square. Orthogonal Latin squares were used for testing compilers Mandl (1985). Orthogonal Latin squares were also used in the testing of network interfaces Williams and Probert (1996).

2.3.2 Orthogonal Array

An Orthogonal Array (OA) is an n*k matrix with run size n, factor number k, and strength t that is denoted by (n, s_i, t). Each column i has exactly s_i symbols, $1 \leq$ i $\leq$ k. In every n*k sub-array, each ordered combination of symbols from the t columns appears equally often in the rows Zhang *et al.* (2014). An OA is simple if it does not contain any repeated rows Wikipedia (2014f). An example of a 2-(4, 5, 1) orthogonal array with a strength 2, and 4 level design of index 1 with 16 runs



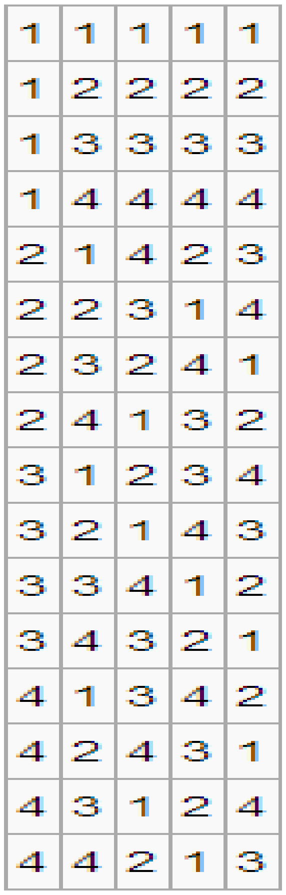
A	B	C	D	E	F	G
B	C	D	E	F	G	A
C	D	E	F	G	A	B
D	E	F	G	A	B	C
E	F	G	A	B	C	D
F	G	A	B	C	D	E
G	A	B	C	D	E	F

Figure 2.1: Latin Square Example

is shown in Figure 2.2 Wikipedia (2014f). An even distribution of all the pairwise combinations of values can be got in any two columns in the array. Orthogonal Array Testing System (OATS), that contains Robust Testing concept, uses orthogonal arrays to generate test suites for a software system Brownlie *et al.* (1992).

2.3.3 Covering Array

A Covering Array (CA) is an n*k array with run size n, factor number k, and strength t denoted by (n, d_i, t) that is similar as OA. Exactly d_i symbols are in each column i, $1 \leq i \leq k$. Each ordered combination of symbols from the t columns appears at least once in every n*k sub-array Zhang *et al.* (2014). For example, a CA with notation (9, 2^4, 3) is shown in Figure 2.3 (a) Ahmed and Zamli (2011). There are four parameters and each one has two values that are represented in nine rows. A mixed level covering array (MCA) denoted by (n, t, k, (v_1, ..., v_k)) is also an n*k array in which the entries of the ith column arise from an alphabet of size v_i; in addition, choosing any t distinct columns i_1, ..., i_t, every t-tuple containing, for $1 \leq j \leq t$, one of the v_{i_j} entries of column i_j, appears in columns i_1, ..., i_t, in at least one of the N rows Colbourn *et al.* (2006). Figure 2.3 (b) represents a MCA with notation (12, 3,



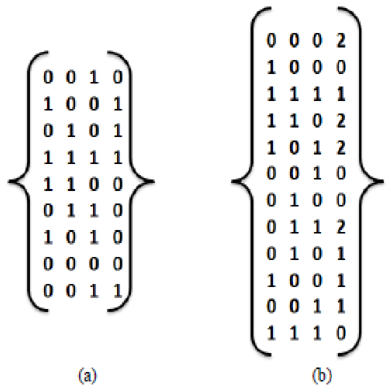
1	1	1	1	1
1	2	2	2	2
1	3	3	3	3
1	4	4	4	4
2	1	4	2	3
2	2	3	1	4
2	3	2	4	1
2	4	1	3	2
3	1	2	3	4
3	2	1	4	3
3	3	4	1	2
3	4	3	2	1
4	1	3	4	2
4	2	4	3	1
4	3	1	2	4
4	4	2	1	3

Figure 2.2: Orthogonal Array Example

2^3, 3^1) Ahmed and Zamli (2011). There are four parameters having three values and five parameters having four values to cover 4-way interactions that are represented in 12 rows.

2.4 Combinatorial Testing

A large number of components are used in software development. Faults often arise from unexpected interactions among the components during software execution Zhang *et al.* (2014). Combinatorial Testing (CT) is type of software testing methods in revealing these faults. It tests all possible discrete combinations of input parameters Wikipedia (2014a). CT can detect failures triggered by interactions of parameters



$$\text{(a)}\ \left\{\begin{matrix} 0&0&1&0\\ 1&0&0&1\\ 0&1&0&1\\ 1&1&1&1\\ 1&1&0&0\\ 0&1&1&0\\ 1&0&1&0\\ 0&0&0&0\\ 0&0&1&1 \end{matrix}\right\} \qquad \text{(b)}\ \left\{\begin{matrix} 0&0&0&2\\ 1&0&0&0\\ 1&1&1&1\\ 1&1&0&2\\ 1&0&1&2\\ 0&0&1&0\\ 0&1&0&0\\ 0&1&1&2\\ 0&1&0&1\\ 1&0&0&1\\ 0&0&1&1\\ 1&1&1&0 \end{matrix}\right\}$$

Figure 2.3: CA and MCA Examples

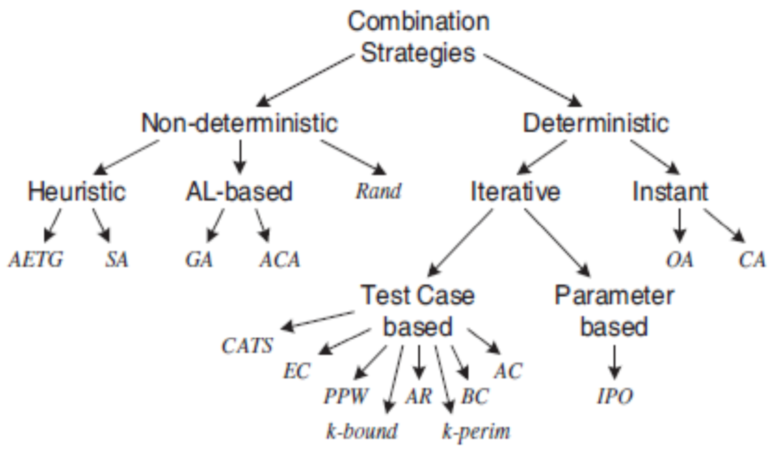


Figure 2.4: Classification Scheme for Combination Strategies Grindal *et al.* (2005a)

with a covering array test suite generated by some sampling mechanisms. Different CT strategies are shown in Figure 2.4. There are two main types of CT strategies. One is deterministic and the other one is non-deterministic.

As the number of possible combinations is too large, CT needs to use a relatively small number of test suites to cover as many combinations of parameters or conditions as possible. Test coverage measures the amount of testing performed by a set of test and is used to evaluate the efficiency of testing methods.

$$test\ coverage = \frac{number\ of\ coverage\ items\ exercised}{total\ number\ of\ coverage\ items} * 100\%$$

Existing CT methods focus on test coverage and try to use the minimum test cases to reach the highest test coverage. The well-known CT algorithms are briefly discussed in the following paragraphs.

2.4.1 Covering Array for Testing

A CA of strength t is a collection of tests so that every t-way interaction is covered by at least one of the tests. CAs reveal faults that arise from improper interaction of t or fewer elements Porter *et al.* (2007). The strength of CA is important for testing. The strength t is the set of (P_i, t_i), P_i is a set of parameters, and t_i is a covering strength on P_i, for $1 \leq \mathrm{i} \leq \mathrm{l}$ Zhang *et al.* (2014). (P_i, t_i) covers all t_i-way combinations of P_i. When the strength increases, the number of test cases may increase rapidly and the testing will be more complete Zhang *et al.* (2014). There are numerous computational and mathematical approaches for construction of CAs with few tests Colbourn (2011); Kuliamin and Petukhov (2011).

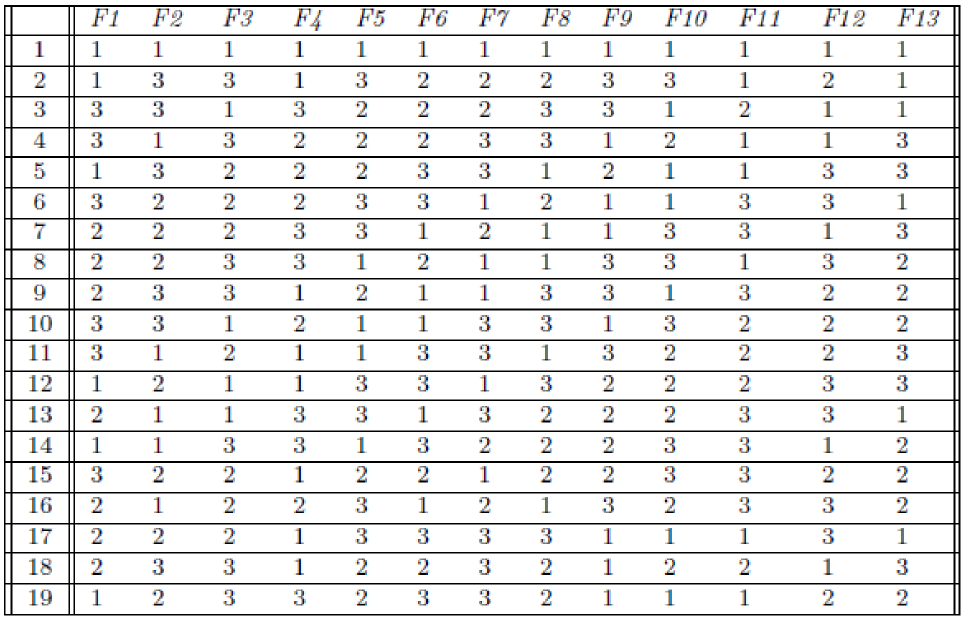
If a t-way interaction causes a fault, executing a test that contains that t-way interaction must reveal the presence of at least one faulty interaction. CAs strive to certify the absence of faults, and are not directed toward finding faults that are present. Executing each test of a CA, certain interactions are then 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 Yilmaz *et al.* (2004), empirical results demonstrate the effectiveness of this strategy at limiting the possible faulty interactions to a manageable number.

2.4.2 Automatic Efficient Test Generator

Combinatorial interaction testing (*CIT*) ensures that every interaction among t or fewer elements is tested, for a specified *strength* t. Among the early methods, Automatic Efficient Test Generator (*AETG*) Cohen *et al.* (1997, 1996a) popularized greedy one-test-at-a-time methods for constructing such 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.

For example, there are 13 components and each component has 3 options (marked as 1, 2, and 3). It would have $3^{13} = 1{,}594{,}323$ test cases. All pairwise interactions can be checked with the 19 test cases shown in Figure 2.5 Cohen *et al.* (1994). This is a reduction of more than 99.999% from the 1,594,323 tests required for exhaustive testing.

Another way to evaluate combination strategies is on the basis of achieved code coverage of the generated test suites Grindal *et al.* (2005a). Test suites generated by AETG for 2-wise coverage reached over 90% block coverage Cohen *et al.* (1996b). AETG reached 93% block coverage with 47 test cases, compared with 85% block coverage for a restricted version of Base Choice (BC) using 72 test cases Burr and Young (1998).



	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	3	3	1	3	2	2	2	3	3	1	2	1
3	3	3	1	3	2	2	2	3	3	1	2	1	1
4	3	1	3	2	2	2	3	3	1	2	1	1	3
5	1	3	2	2	2	3	3	1	2	1	1	3	3
6	3	2	2	2	3	3	1	2	1	1	3	3	1
7	2	2	2	3	3	1	2	1	1	3	3	1	3
8	2	2	3	3	1	2	1	1	3	3	1	3	2
9	2	3	3	1	2	1	1	3	3	1	3	2	2
10	3	3	1	2	1	1	3	3	1	3	2	2	2
11	3	1	2	1	1	3	3	1	3	2	2	2	3
12	1	2	1	1	3	3	1	3	2	2	2	3	3
13	2	1	1	3	3	1	3	2	2	2	3	3	1
14	1	1	3	3	1	3	2	2	2	3	3	1	2
15	3	2	2	1	2	2	1	2	2	3	3	2	2
16	2	1	2	2	3	1	2	1	3	2	3	3	2
17	2	2	2	1	3	3	3	3	1	1	1	3	1
18	2	3	3	1	2	2	3	2	1	2	2	1	3
19	1	2	3	3	2	3	3	2	1	1	1	2	2

Figure 2.5: AETG Example Cohen *et al.* (1994)

2.4.3 In-Parameter-Order

The in-parameter-order (IPO), as one greedy strategy of generating CAs, was proposed by Lei and Tai to extend CA in parameter order for combinatorial testing Lei and Tai (1998); Zhang *et al.* (2014). The extension process starts from a pairwise test set generated for the first two parameters. It gradually extends a small CA to a large CA by adding one additional parameters each time. When an additional parameter is added, the existing pairwise test set extends in horizontal and vertical direction respectively Lei and Tai (1998).

- *Horizontal extension:* Add a new column, when a new parameter is added.
- *Vertical extension:* Add new rows to cover those uncovered combinations by horizontal extension.

The extension process repeats until all parameters are covered.

For instance, a system has three parameters A, B, and C Lei (2005).

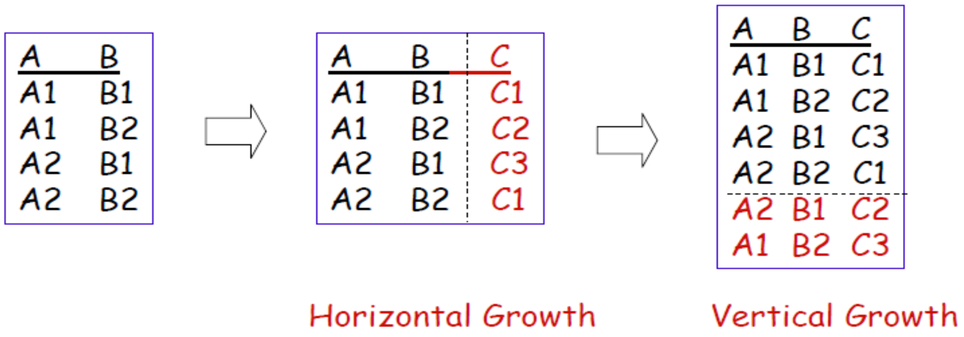


Figure 2.6: IPO Example Lei (2005)

- Parameter A has values A1 and A2;
- Parameter B has values B1 and B2;
- Parameter C has values C1, C2, and C3.

The IPO extension process is shown in Figure 2.6. When parameter C is added, a new column is added for the extension of parameter C. After that, two rows are added according to the extension of parameter C.

The time complexity of IPO is superior to the time complexity of AETG Grindal *et al.* (2005a). IPO has a time complexity of $O(v^3N^2\log(N))$ and AETG has a time complexity of $O(v^4N^2\log(N))$, where N is the number of parameters, each of which has v values Tai and Lei (2002).

2.4.4 Genetic Algorithm

A genetic algorithm (GA) is a search heuristic that mimics the process of natural selection Wikipedia (2014d). GA is best defined as a pollution based search algorithm based loosely on concepts from biologic evolution Rajappa *et al.* (2008). GA is an iterative algorithm that is used to find CAs. In each iteration, it involves inheritance, mutation, selection, and crossover. A chromosome as a candidate solution that is distinct pairwise interaction covered by its configuration is evaluated by GA Ghazi and Ahmed (2003). The basic AETG is extended with GA. The uncovered new t-way combinations are covered by AETG-GA. In each generation, the best chromosomes are kept and survive to the next generation Shiba *et al.* (2004).

2.4.5 Backtracking Algorithm

Backtracking algorithm is used to finding solutions of constraint satisfaction problems and is often implemented by a search tree. It extends a partial solution by choosing values for variables incrementally until all constraint are satisfied Zhang *et al.* (2014). It abandons each partial solution as soon as it determines that the partial solution cannot possibly be completed to a valid solution Wikipedia (2014b). Unlike brute force, backtracking checks candidate solutions, if any constraint is violated, when a variable is assigned Zhang *et al.* (2014).

2.4.6 Fault Detection

There are conflicting claims in the literature concerning the effectiveness of random, anti-random, and combinatorial interaction test suites at finding faults. According to Huang *et al.* (2012), ART-based tests cover all t-way interactions more quickly than randomly chosen tests. At the same time they often detect more failures earlier and with fewer test cases. According to Cohen *et al.* (1997); Dalal *et al.* (1999); Yilmaz *et al.* (2004), combinatorial interaction testing yields small test suites with high code coverage and good fault detection ability. In CIT, construction of the best test suite Grindal *et al.* (2005b); Nie and Leung (2011) can be costly; even a solution with a small number of tests that guarantees complete coverage of t-way interactions may be difficult to produce. This has led to the frequent use of random testing Arcuri *et al.* (2010); Duran and Ntafos (1984).

Schroeder *et al.* Schroeder *et al.* (2004) compare the fault detection effectiveness of combinatorial interaction test suites with equally-sized random test suites. Their results indicate that there is no significant difference in the fault detection effectiveness. Dalal and Mallows Dalal and Mallows (1998) also indicate that no matter the input size is, the numbers of interactions covered in same-sized random and combinatorial interaction test suites are similar in many cases. However, Bryce and Colbourn Bryce and Colbourn (2007, pear) observe that these comparisons used covering arrays that, while the best known at the time, are far from the smallest ones available. Repeating the determination of fault detection times using the smaller arrays now known changes the conclusion completely. Indeed for the situations examined in Dalal and Mallows (1998); Schroeder *et al.* (2004), improving the size of the covering array used results in the random method covering a much smaller fraction of the possible faults. Moreover, covering arrays generated by a one-test-at-a-time method produced the best rate of fault detection.

2.5 Adaptive Reasoning Algorithm

The Adaptive Reasoning (AR) algorithm is a strategy to detect faults in SaaS Tsai *et al.* (2013b). 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 configuration fails a test, there is at least one faulty interaction covered by the tenant configuration.
- **Principle 2:** When a tenant application passes a test, there is no faulty interaction covered by the tenant configuration.
- **Principle 3:** Whenever a configuration covers one or more faulty interactions, it is faulty.

2.5.1 Other Related Topics and Proposal Motivation

Test results are used to isolate the faulty combinations that cause the software under test to fail. Effective classification can increase efficiency Shakya *et al.* (2012): The faulty combinations in scenarios where failures are not commonly observed are classified. Test augmentation and feature selection can be used to enhance classification.

ACTS (Advanced Combinatorial Testing System), a combinatorial test generation research tool, supports t-way combinatorial test generation with several advanced features such as mixed-strength test generation and constraint handling Yu *et al.* (2013); Borazjany *et al.* (2012).

Existing CT methods use different strategies to generate test cases and only identify faulty configurations, but do not exploit the faulty root of each configuration. Our methods do not rely on whether random, anti-random, combinatorial interaction, or another type of combination-based test suite generation is used. We focus on the task of large-scale distributed testing, analyzing, merging and maintaining test results in order to reduce the amount of testing needed.

Chapter 3

TEST ALGEBRA FOR CONCURRENT COMBINATORIAL TESTING

3.1 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 a passing 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`):** 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 none 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 the 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.

3.1.1 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 $\{\mathtt{X}, \mathtt{F}, \mathtt{P}, \mathtt{N}, \mathtt{U}\}$ 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

The binary operation $\otimes$ is commutative and associative (see the appendix for proofs):

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

Using the definition of $\otimes$, $V(\mathcal{T}_1 \cup \mathcal{T}_2) \succeq V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)$. 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 $V(S) = \mathtt{X}$ but whenever $S' \subset S$, $V(S') \prec \mathtt{X}$. The X-*implicants* provide a compact representation for all interactions that are infeasible. Indeed for any interaction $\mathcal{T}$ that contains an X-*implicant*, $V(\mathcal{T}) = \mathtt{X}$. Furthermore, a set S is an F-*implicant* if $V(S) = \mathtt{F}$ but whenever $S' \subset S$, $V(S') \prec \mathtt{F}$. For any interaction $\mathcal{T}$ that contains an F-*implicant*, $V(\mathcal{T}) \succeq \mathtt{F}$. In the same way, a set S is an N-*implicant* if $V(S) = \mathtt{N}$ but whenever $S' \subset S$, $V(S') = \mathtt{U}$. For any interaction $\mathcal{T}$ that contains an N-*implicant*, $V(\mathcal{T}) \succeq \mathtt{N}$. An analogous statement holds for passed interactions, but here the implication is for subsets. A set S is a $\texttt{P}$-*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 $\texttt{P}$-*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 $\texttt{X}$-, $\texttt{F}$-, or $\texttt{N}$-*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 $\texttt{P}$-*implicant*, and every $(t+1)$ interaction contained in one of these interactions can be assigned status $\texttt{P}$.

For example, suppose that $V(a,b) = \texttt{P}$ and $V(a,e) = \texttt{X}$. Then $V(a,b,e) \succeq V(a,b) \otimes V(a,e) = \texttt{X}$.

The valuation of the 3-way interaction (a,b,c) can often be inferred from valuations 2-way interactions (a,b), (a,c), (b,c). If any contained 2-way interaction has value $\texttt{F}$, the valuation of (a,b,c) is $\texttt{F}$, without further testing needed. But if all values of the contained 2-way interactions are $\texttt{P}$, then (a,b,c) either has valuation $\texttt{X}$ or $\texttt{N}$, or it needs to be tested (its valuation is currently $\texttt{U}$).

3.1.2 Changing Test Result Status

The status of a configuration is determined by the status of all interactions that it covers.

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 $\mathcal{T}$.

3.1.3 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.

$\cup$	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)

(a) Union of all Possible Interactions

$\otimes$	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)											

(b) $\otimes$ Operation of all Possible Interactions

Figure 3.1: 3-way Interaction Operation Table

As shown in Figure 3.1, 3-way interactions can be composed by using 2-way interactions and components. Thus, following the TA implication rules, the 3-way interaction 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 3.1, 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 the TA defines the order of the five status indicators, the result is the highest obtained. 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						

3.1.4 Relationship Between Configuration and Its Interactions

One configuration contains many different interactions. The status of one configuration is composed by merging tests results of all its interactions. The status of $\mathcal{T}$ can be defined as $V(\mathcal{T}) = \bigodot_{\mathcal{I} \subseteq \mathcal{T}} V(\mathcal{I})$, where $\mathcal{I}$ is an interaction covered by configuration $\mathcal{T}$ and $\odot$ is defined as

$\odot$	X	F	P	N	U
X	X	X	X	X	X
F	X	F	F	F	F
P	X	F	P	U	U
N	X	F	U	U	U
U	X	F	U	U	U

That is

1. If any interaction covered by configuration $\mathcal{T}$ has a status X, then $V(\mathcal{T}) = \texttt{X}$; (Otherwise, at least one interaction of configuration $\mathcal{T}$ is allowed by the specification, it cannot say that $\mathcal{T}$ is infeasible.)

2. If any interaction covered by configuration $\mathcal{T}$ has a status `F` and no one is infeasible, then $V(\mathcal{T}) = \texttt{F}$;

3. If all interactions of configuration $\mathcal{T}$ are irrelevant or unknown, then $V(\mathcal{T}) = \texttt{U}$;

4. If some interactions covered by configuration $\mathcal{T}$ have status `P`, the other ones have status `N` or `U`, and no one is infeasible or failed, then $V(\mathcal{T}) = \texttt{U}$, so further testing is needed to determine the status of configuration $\mathcal{T}$;

5. All interactions covered by configuration $\mathcal{T}$ have status `P`, then $V(\mathcal{T}) = \texttt{P}$.

For example, suppose that configuration $\mathcal{T}$ has three interactions $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$. According to $\mathcal{T} = (\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3)$, combinations of interaction test results can be used to determine the configuration test result. Based on the TA associative rules,

$$V(\mathcal{T}) = V(\mathcal{I}_1) \odot V(\mathcal{I}_2) \odot V(\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_2) \odot V(\mathcal{I}_2,\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_2,\mathcal{I}_3)$$
$$= (V(\mathcal{I}_1) \odot V(\mathcal{I}_2) \odot V(\mathcal{I}_3)) \odot (V(\mathcal{I}_1,\mathcal{I}_2) \odot V(\mathcal{I}_2,\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_3)) \odot V(\mathcal{I}_1,\mathcal{I}_2,\mathcal{I}_3)$$
$$= (V(\mathcal{I}_1) \odot V(\mathcal{I}_2) \odot V(\mathcal{I}_3)) \odot (V(\mathcal{I}_1,\mathcal{I}_2) \odot V(\mathcal{I}_2,\mathcal{I}_3)) \odot (V(\mathcal{I}_1,\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_2,\mathcal{I}_3))$$
$$= (V(\mathcal{I}_1) \odot V(\mathcal{I}_2)) \odot (V(\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_2)) \odot (V(\mathcal{I}_2,\mathcal{I}_3) \odot V(\mathcal{I}_1,\mathcal{I}_3)) \odot V(\mathcal{I}_1,\mathcal{I}_2,\mathcal{I}_3)$$
$$= \ldots\ldots$$

3.1.5 Merging Concurrent Testing Results

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

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

$\oplus$	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

Operation $\oplus$ is also commutative and associative; see the appendix for proofs.

Using $\oplus$, 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 $\oplus$. If the constraints of the merge are satisfied and $V(\mathcal{T}) = \texttt{X}$, F, or P, the results can only be changed when there are errors in testing. When $V(\mathcal{T}) = \texttt{E}$, the testing environment must be corrected and tests executed again after fixing the error(s) in testing. For example, when

$V_{\text{cluster1}}(a,c,e) = \mathtt{X}$ and $V_{\text{cluster2}}(a,c,e) = \mathtt{F}$, $V_{\text{merged}}(a,c,e) = \mathtt{X} \oplus \mathtt{F} = \mathtt{E}$. The error with the tests for interaction (a,c,e) must be fixed.

Using associativity of $\oplus$,

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

Thus one can partition the configurations into overlapping sets for different servers. Conventional cloud computing operations such as MapReduce require that data should not overlap. In TA, this is not a concern.

There are six components $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, $\mathcal{T}_4$, $\mathcal{T}_5$, $\mathcal{T}_6$. They have the following relationships. $\mathcal{T}_1 = \mathcal{T}_2 \cup \mathcal{T}_3$, $\mathcal{T}_2 = \mathcal{T}_4 \cup \mathcal{T}_6$, $\mathcal{T}_3 = \mathcal{T}_4 \cup \mathcal{T}_5$.

- $V(\mathcal{T}_1) = V(\mathcal{T}_2 \cup \mathcal{T}_3) = V(\mathcal{T}_2) \otimes V(\mathcal{T}_3) = (V(\mathcal{T}_4) \otimes V(\mathcal{T}_6)) \otimes (V(\mathcal{T}_4) \otimes V(\mathcal{T}_5))$ $= (V(\mathcal{T}_4) \otimes V(\mathcal{T}_4)) \otimes (V(\mathcal{T}_6) \otimes V(\mathcal{T}_5)) = V(\mathcal{T}_4) \otimes V(\mathcal{T}_6) \otimes V(\mathcal{T}_5)$
- $V_1(\mathcal{T}_1) \oplus V_2(\mathcal{T}_1) = V_1(\mathcal{T}_2 \cup \mathcal{T}_3) \oplus V_2(\mathcal{T}_2 \cup \mathcal{T}_3) = (V_1(\mathcal{T}_2) \otimes V_1(\mathcal{T}_3)) \oplus (V_2(\mathcal{T}_2) \otimes V_2(\mathcal{T}_3))$
- $V_x = V_1(\mathcal{T}_1) \oplus V_3(\mathcal{T}_1) = V_y = V_1(\mathcal{T}_1) \oplus V_2(\mathcal{T}_1)$

In last paragraph, it uses $\odot$ to analyze the relationship between configuration and its interactions. For example, configuration $\mathcal{T}$ has four interactions $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, $\mathcal{I}_4$. Three servers ($server_1$, $server_2$, $server_3$) are used to test these interactions. Test workloads may be assigned to different servers. The returned test results may be overlapping.

	$Server_1$	$Server_2$	$Server_3$
$\mathcal{I}_1$	*assigned*	*assigned*	
$\mathcal{I}_2$		*assigned*	*assigned*
$\mathcal{I}_3$	*assigned*	*assigned*	
$\mathcal{I}_4$	*assigned*	*assigned*	*assigned*

$\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, and $\mathcal{I}_4$ have 2, 2, 2, and 3 test results from different servers respectively. Total 24 possible combinations of interaction test results can be used to finalize configuration test result through dot operation defined by TA. During using $\odot$ operation, $\oplus$ can also be used to merge results from different servers. The following results can be derived according to $\oplus$ and $\odot$ rules.

$$V(\mathcal{T}) = V_{s_1}(\mathcal{I}_1) \odot V_{s_2}(\mathcal{I}_2) \odot V_{s_1}(\mathcal{I}_3) \odot V_{s_1}(\mathcal{I}_4)$$

$$= V_{s_1}(\mathcal{I}_1) \odot V_{s_3}(\mathcal{I}_2) \odot V_{s_2}(\mathcal{I}_3) \odot V_{s_1}(\mathcal{I}_4)$$

$$= V_{s_2}(\mathcal{I}_1) \odot V_{s_3}(\mathcal{I}_2) \odot V_{s_2}(\mathcal{I}_3) \odot V_{s_2}(\mathcal{I}_4)$$

$$= V_{s_2}(\mathcal{I}_1) \odot V_{s_3}(\mathcal{I}_2) \odot V_{s_1}(\mathcal{I}_3) \odot V_{s_3}(\mathcal{I}_4)$$

$$= (V_{s_1}(\mathcal{I}_1) \oplus V_{s_2}(\mathcal{I}_1)) \odot V_{s_3}(\mathcal{I}_2) \odot V_{s_2}(\mathcal{I}_3) \odot V_{s_1}(\mathcal{I}_4)$$

$$= (V_{s_1}(\mathcal{I}_1) \oplus V_{s_2}(\mathcal{I}_1)) \odot (V_{s_2}(\mathcal{I}_2) \oplus V_{s_3}(\mathcal{I}_2)) \odot V_{s_2}(\mathcal{I}_3) \odot V_{s_1}(\mathcal{I}_4) = (V_{s_1}(\mathcal{I}_1) \oplus V_{s_2}(\mathcal{I}_1)) \odot (V_{s_2}(\mathcal{I}_2) \oplus V_{s_3}(\mathcal{I}_2)) \odot (V_{s_1}(\mathcal{I}_3) \oplus V_{s_2}(\mathcal{I}_3)) \odot (V_{s_1}(\mathcal{I}_4) \oplus V_{s_2}(\mathcal{I}_4) \oplus V_{s_3}(\mathcal{I}_4))$$

$$= ......$$

3.1.6 Distributive Rule

To examine the distributivity of $\otimes$ over $\oplus$, the definition of $\otimes$ is extended to support `E`:

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

In general, the distributivity of $\otimes$ over $\oplus$ does not hold. Instead, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) \succeq (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2))$.

Equality holds when $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is not `E` (this distributivity of $\otimes$ over $\oplus$ is proved in appendix). This can be used to further merge concurrent testing results. For example, test result of configuration (a, b, c) is infeasible that is merged by three test results from $Server_1$, $Server_2$, and $Server_3$. Similarly, configuration (b, c, d) has status `P` that is merged by two test results from $Server_1$ and $Server_2$. Configuration (a, b, c, d) contains configuration (a, b, c), (b, c, d), and other configurations. The test result for configuration (a, b, c, d) can be obtained by merging test results of all of these configurations. Since $V(a, b, c) = \texttt{X}$, no matter what status of other configuration is, configuration (a, b, c, d) always has infeasible status.

3.1.7 Relationship Among Different Type Configurations

This paragraph explores the relationship among different type configurations. The definition of each factor:

- *Component: E*
- *Options of Each Component: K*
- *Configuration: G*
- *Infeasible Configuration: X*
- *Faulty Configuration: F*
- *Operational Configuration: P*
- *Irrelevant Configuration: N*
- *Unknown Configuration: U*
- *Repeating Counted Configuration: R*

The total number of configurations (G) is $(2K)^E$. TA is only used to analyze t-way configurations for $2 \leq t \leq 6$ in this paper. The following equations show the number of each type configuration.

- *Number of Configurations (NC):*

  $\sum_{m=2}^{6} C_E^m K^m$, $m \in Z$

- *Number of Infeasible Configurations (NXC):*

  $\sum_{m=2}^{6}[X_m(1 + \sum_{n=1}^{6-m} C_{E-m}^n)] - R_X$, $m \in Z$, $n \in Z$

- *Number of Faulty Configurations (NFC):*

  $\sum_{m=2}^{6}[F_m(1 + \sum_{n=1}^{6-m} C_{E-m}^n)] - R_F$, $m \in Z$, $n \in Z$

- *Number of Operational Configurations (NPC):*

  $\sum_{m=2}^{6} P_m$, $m \in Z$

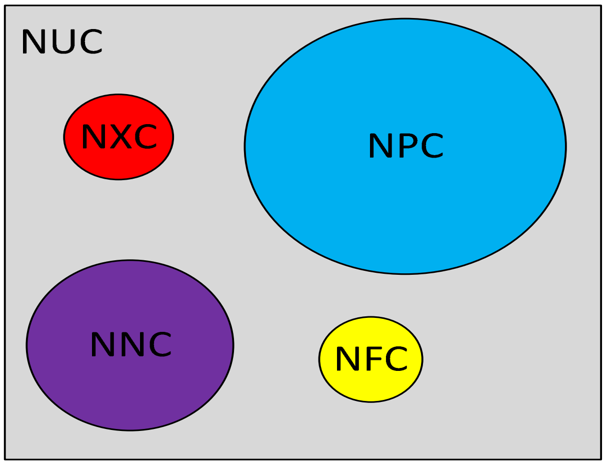


Figure 3.2: The Number of Configurations

- *Number of Irrelevant Configurations (NNC):*

  $\sum_{m=2}^{6} N_m,\ m \in Z$

- *Number of Unknown Configurations (NUC):*

  $\sum_{m=2}^{6} U_m = NUC = NC - NXC - NFC - NPC - NNC,\ m \in Z,\ n \in Z$

The relationships among different type configuration are shown in Figure 5.1. The rectangle is the the number of configurations. Infeasible, faulty, operational, and irrelevant configuration is represented by red, yellow, blue, and purple circle respectively. The remain grey part of rectangle represent unknown configurations. The size of each circle corresponds to the number of related configurations. Operational and irrelevant configurations have larger number than infeasible and faulty configurations.

Since infeasible, faulty, and irrelevant configurations can be used to reduce the TA analysis workloads, it gets the following conclusions.

- If the number of infeasible (`X`) configurations is large or increases, more related infeasible configurations will be eliminated for testing consideration.

- If the number of faulty (`F`) configurations is large or increases, more related faulty configurations will be eliminated for testing consideration.

- If the number of irrelevant (`N`) configurations is large or increases, more related irrelevant configurations will be eliminated for testing consideration.

Those infeasible, faulty, and irrelevant configurations can be used to eliminate candidate testing configurations. Oppositely, operational configurations do not work for eliminating candidate testing configurations. The operational configurations can be used as subsets of other candidate configurations. Their operational results can be reused in TA analysis. So maximizing the infeasible, faulty, and irrelevant configurations as the initial settings is a good way to eliminate related configurations in testing consideration and increase the testing efficiency. However, the meta infeasible and faulty configurations must be got though testing, so it is difficult to maximize infeasible and faulty configurations at the beginning of testing. But according to the relationships among different components, it is easy to maximize irrelevant configurations in a short time.

myexperiment.org showed that a lot of `N` can be generated or the number of `N` can be very large. So large, `N` table may not be necessary, anything not in `X`, `F`, `P`, `U` tables can be considered as in the `N` table.

While the number of possible combinations of software published at myexperiment.org is large, but most of these combinations are `N` (irrelevant) and thus do not need to be tested. The following examples illustrate this point.

- *QR code (matrix code) generator*

  This workflow uses the QR code service provided by the ChemTools project. It has four components (shown in Table 3.1), the total number of possible combinations is 16. The total number of actual combinations is 2. Actual combinations are only 12.5% of possible combinations. In this example, 12.5% of possible combinations needs to be analyzed. The other 87.5% of possible

Table 3.1: QR Code Generator Components

	Type	**Number**
1	Input	1
2	QR code	1
3	Output	2

Table 3.2: Entrez Gene to KEGG Pathway Components

	Type	**Number**
1	Input	2
2	Add, split	2
3	Gene	3
4	Merge	2
5	Output	2

combinations are irrelevant that can be excluded from TA analysis.

- *Entrez Gene to KEGG Pathway*

  This workflow takes in Entrez gene ids then adds the string ncbi-geneid: to the start of each gene id. These gene IDs are then cross-referenced to KEGG gene IDs. Each KEGG gene ID is then sent to the KEGG pathway database and its relevant pathways returned. It has eleven components (shown in Table 3.2), and the total number of possible combinations is 2048. The total number of actual combinations is one. Actual combinations are only 0.0488% of possible combinations.

- *Gene annotation pipeline for graves disease scenarios illustrates the feasibility of combinatorial testing for large component size*

  This is a revised workflow for the Graves disease scenario gene annotation

Table 3.3: Gene Annotation Pipeline Components

	Type	**Number**
1	Input	1
2	Get data	13
3	Get string and diagram	2
4	Remove and clean data	4
5	Calculation	5
6	Other	1
7	Output	12

pipeline used in the myGrid project. The workflow had to be re-written due to the loss of the services invoked in the original workflow. It has 48 components (shown in Table 3.3), and the total number of possible combinations is 2^{48}. The total number of actual combinations is 19. Actual combinations are only $6.75 * 10^{-12}\%$ of possible combinations. Almost all possible combinations are irrelevant that can be excluded from TA analysis.

3.1.8 Measurements

- *Related Configurations (RC):*

For example, if the status of a configuration $C = (C_1, C_2, C_3)$ is known, significant information is about those configuration D that contain C as a sub-configuration.

  - Related Configurations or $RC(t, C)$ = the set of configurations that have t elements with C as a sub-configuration.
  - Cumulative Related Configurations or $CRC(w, C) = Union(RC(t, C))$ for any $t \leq w$, w is the total number of related configurations.

  - $CRC(w, C) \geq RC(t, C)$, if $w \geq t$.

- *Reduction Ratio (RR):*
  - If the number of configurations needs to be tested after TA analysis is NC. $RR = NC/total\ number\ of\ configurations$.
  - `X`, `F`, `N` can reduce the number of configurations that need to be tested by generating $CRC(6, sum(\mathtt{X}, \mathtt{F}, \mathtt{N}))$. $NC = CRC(6, sum(\mathtt{X}, \mathtt{F}, \mathtt{N}))$, and it is computed by the following TA analysis algorithm:
    - ∗ Starting from 2-way configurations, the simulation goes through each 2-way configuration, eliminate those initial `X`, `F`, `P`, and `N` 2-way configurations, and identify those needs to be tested. For those needs to be test, obtain the simulated test results.
    - ∗ Once completing the 2-way configurations, all the tables are updated, and start identifying those 3-way configurations that need to be tested using the TA rules.
    - ∗ Once completing the 3-way configurations, all the tables are updated, and the process repeats for 4-way configurations to 6-way configurations.

    This can be done by TA computation, but TA analysis followed by testing, followed by TA analysis, from $t = 2$ to $t = 6$.
  - Parallel computing can be used to increase the speedup of TA analysis and enhance the efficiency of TA analysis.

3.1.9 Incremental Development

Starting from a small SaaS, or a small subset of a SaaS, testing the tenant applications, and gradually develop the `X`, `F`, `P`, and `U` table incrementally.

- When a new component arrives without any association with any tenant, all configurations with any components will be marked `N`.
- When a new component arrives and associated with a set of tenant applications, all configurations of this component with any 2-way to 6-way configurations will be marked as `U`, and sent to testing and TA analysis.
- TA will analyze if any of these new tenant applications need not be tested (if any of them contains any `X`, `F`, or `N`).
- When a tenant application is tested, all 2-way, 3-way,.. 6-way configurations with any new components will be marked `P`.
- If the tenant application is faulty, the faulty interaction must be identified, one possible algorithm is to use AR algorithm to do so.

3.2 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.

Chapter 4

CONCURRENT TEST ALGEBRA EXECUTION WITH COMBINATORIAL TESTING

4.1 TA Analysis Framework

Figure 4.1 shows the relationship between TA and combinatorial testing. Combinatorial testing can use AETG, AR, or IPO Calvagna and Gargantini (2009) to identify `P` and `F` configurations, and even fault locations. The identified configurations and fault locations are saved as test results for future use. TA automatically detects `X` or `F` configurations using existing `X` or `F` results. All `X` and `F` configurations are eliminated from testing considerations. In combinatorial testing, test workloads of those `X` and `F` candidate configurations are reduced by TA analysis. Similarly, those `N` configurations can also be eliminated from considerations.

Figure 4.2 shows the concurrent design for TA analysis. There are many candidate components for tenants to pick up to compose their own applications. The composed applications of different tenants will be assigned to different clusters for analysis.

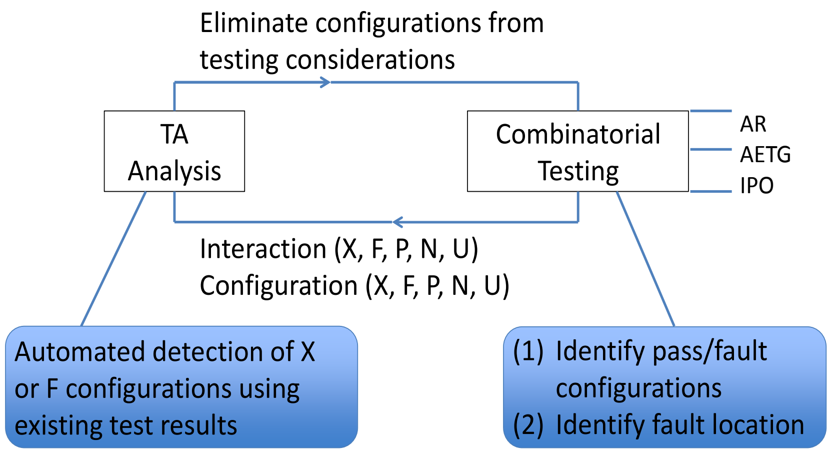

Eliminate configurations from testing considerations
TA Analysis
Combinatorial Testing
AR
AETG
IPO
Interaction (X, F, P, N, U)
Configuration (X, F, P, N, U)
Automated detection of X or F configurations using existing test results
(1) Identify pass/fault configurations
(2) Identify fault location

Figure 4.1: The Relationship Between TA and Combinatorial Testing

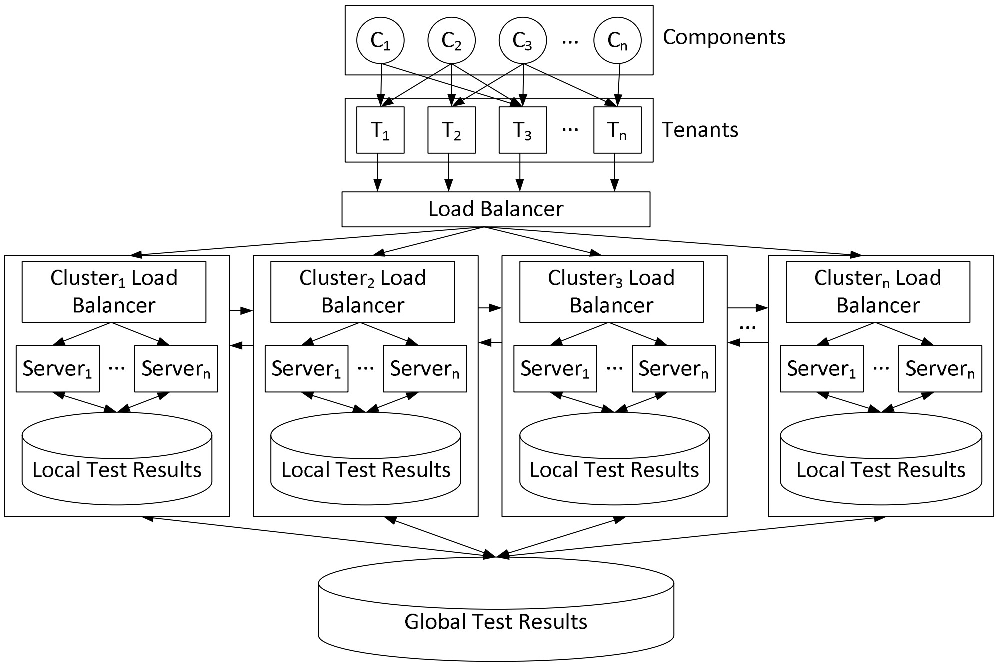


Figure 4.2: Concurrent TA Analysis Design

Each cluster has multiple servers to handle TA tasks in parallel.

The two-level architecture not only automatically balances the workloads across multiple clusters and servers, but also scales up with increasing loads with automated expansion. This is similar to the scalability architecture commonly used in SaaS Tsai *et al.* (2012).

- Allocate by tenants at the first level: The tenants will be clustered based on the similarity measured by configurations. Two tenants are similar to each other if two share many components. similar tenants are grouped and assigned to the same cluster.

- Allocate by configurations at second level: The configurations of each tenant assigned to one cluster will be assigned to the same or different server in each cluster for analyzing.

Concurrent algorithm is proposed to solve the distribution and collection of testing workloads.

Algorithm 1 Concurrent for TA Analysis

```
Input:
    Candidate test configuration $c_i$, map cluster $p_i$,
Output:
    Reduce testing result $r_i$, testing result of test case $tr_i$,
1: $p_i$ = (id $n$, $\sum c_i$ of one candidate test set)
2: for all $p_i$ do
3:     return $tr_i$
4: end for
5: $r_i = (n, \sum c_i, tr_i)$
6: for all $p_i$ do
7:     return $r_i$
8: end for
```

The high level load balancer allocates testing and TA analysis tasks to different clusters, and each cluster has its own local load balancer and it will dispatch testing and TA analysis tasks to different servers within the cluster. All clusters share a global database, and each sever within a cluster shares a local database for efficient processing.

Test scripts and databases can be stored at the global database as well as at local databases within clusters. As TA rules automatically detect test result consistency, thus any temporary inconsistency between local database with the global database can be resolved quickly once communicated.

The finished test results will be saved as PTR (Previous Test Result) and shared. Before saving, all test results must be verified by the test oracle to check the correctness. Only the correct test results will be saved in test database. Same configurations may be analyzed in different clusters. In this case, if one cluster gets test result of any configuration first, it can be shared and reused by others. For example, $cluster_1$ gets that configuration (a, b) fails in combinatorial testing first. $cluster_2$ and $cluster_3$ can reuse the shared faulty result of configuration (a, b).

The following example illustrates the testing process of fifteen configurations. All feasible configurations should be tested. For simplicity, assume that only configuration (c, d, f) is faulty, and only configuration (c, d, e) is infeasible, and all other configurations are operational. The existing test results of configurations can be used to analyze test results of candidate configurations for reducing test workloads.

Example 1: If one assigns 1-10 configurations into $Server_1$, 6-15 configurations into $Server_2$, and 1-5, 11-15 configurations into $Server_3$.

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

Example 2: 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 configuration (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.

	$Server_1$	$Server_2$	$Server_3$	$Merged\ Results$
(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

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.

The returned merged results from these two examples are same. Analyzing the returned results, it can get the following results:

- **All 2-way interactions:** All of them pass the testing.
- **All 3-way interactions:** Except interaction (c, d, e) and (c, d, f), all the left 3-way interactions pass the testing.
- **All 4-ways interactions:** The 4-way interactions that contain (c, d, e) and (c, d, f), such as (a, c, d, e), (a, c, d, f), (b, c, d, e), (b, c, d, f), and (c, d, e, f), do not pass the testing. All the left 4-way interactions pass the testing.
- **All fifteen configurations:** Configurations (a, c, d, e), (a, c, d, f), (b, c, d, e), (b, c, d, f), and (c, d, e, f) do not pass the testing. All the left configurations pass the testing.

4.1.1 *The Role of* `N` *in Concurrent Combinatorial Testing*

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 interaction $\mathcal{T}_j$ in the configuration CS_i. If an algorithm such as AR is used, any `P` results reduce the number of tests needed for t-way interaction testing. Any `F` result in TA is useful to use 2-way, 3-way, and $(t-1)$-way interaction testing results to reduce testing effort for t-way interaction testing. Thus, both `P` and `F` results are useful in reducing testing effort. The `N` results are also useful, and any configuration that is marked as `N` means that it is not necessary to perform testing, and this can reduce the number of configurations and interactions to test. This is particularly useful when the number of configurations is large, as the set of configurations can be divided into different (not necessarily non-overlapping) sets, one for each server or cluster of servers. In this way, any `N` results help in divide-and-conquer approach to address large combinatorial testing. For example, if a SaaS system has $1M$ tenant applications, but the SaaS platform has over $10,000$ processors, then each processor needs to handle only $\frac{1M}{10K} = 100$ tenant applications. Each server can divide this testing process again, performing testing 10 times, each testing 10 tenants with test results stored in the database. In this way, large-scale combinatorial testing can be performed.

4.1.2 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 C_2^n. In general, the number of t-way interactions is equal to C_t^n. More interactions are treated when $C_t^n > C_{t-1}^n$, which happens when $t \leq \frac{n}{2}$. The total number of interactions examined is $\sum_{i=2}^{t} C_i^n$.

4.2 TA Analysis Algorithm

4.2.1 Search Process and Algorithm

For new candidate testing configurations, the testing process as follows:

1. **Search in F-table**

   For n-way candidate configuration, search in related F-table from 2-way to n-way to check whether it contains any `F` interactions. If yes, candidate configuration is faulty and can be eliminated from testing. Otherwise, search in P-table to find which test results can be reused.

   - *Best condition:* For a n-way candidate configuration, related 2-way faulty interaction is found in F-table. Stop searching in F-table and return `F` as the test result of candidate configuration.
   - *Worst condition:* All related interactions of n-way candidate configuration are searched in F-table, but none is found in F-table. The n-way candidate configuration cannot be eliminated from the TA analysis.

2. **Search in P-table**

   For n-way candidate configuration, search all its related interactions from n-way to 2-way interaction P-table. All found interactions can be excluded from the candidate testing list. Only those missing interactions need to be tested.

Algorithm 2 F-table Search Algorithm

```
Input:
    F-table, n-way candidate configuration ($n \leq 6$)
Output:
    Test results of candidate configuration
1: Calculate all related interactions of n-way candidate configurations
2: Search related interactions from 2-way to n-way interaction F-table
3: for (i=2; i<=n; ++) do
4:     Traverse i-way's F-table to search related interactions
5:     if any interactions are found then
6:         Return faulty result
7:         Stop
8:     end if
9: end for
```

- *Best condition:* For an n-way candidate configuration, all related interactions are found in P-table, thus the configuration is operational.
- *Worst condition:* Any related interactions cannot be found in P-table, and thus related interactions need to be tested.

Example 1: Search configuration (a,b,d,f). Search the related interaction of configuration (a,b,d,f) in F-table. Find interaction (d,f) is faulty. So $V(a, b, d, f) = F$.

Example 2: Search configuration (a,b,e,f). Search the related interactions of configuration (a,b,e,f) in F-table. No one can be found. Then search all related interactions in P-table. 3-way interaction (a,b,e), (a,b,f), and (b,e,f) are found in 3-way P-table. As TA proved, $V(a, b, e, f) = V(a, b, e) \bigotimes V(a, b, f) = V(a, b, e) \bigotimes V(b, e, f)$ $= V(a, b, f) \bigotimes V(b, e, f)$. Since all 3-way interactions of (a,b,e,f) can be found in

Algorithm 3 Configuration P-table Search Algorithm

```
Input:
    P-table, n-way candidate configuration ($n \leq 6$)
Output:
    Non-found interaction
1: Calculate all related interactions of n-way candidate configuration from n-way to 2-way
2: Put all related interactions into different lists according to component number
3: Search interactions from n-way to 2-way interaction P-table
4: for (i=n; i<2; −−) do
5:     Traverse i-ways P-table to search for i-way interaction of candidate configuration
6:     if any interactions are found then
7:         Delete the found interactions from the list
8:         if the list is empty then
9:             Return empty list
10:            Stop
11:        end if
12:    end if
13:    Return list
14: end for
```

P-table, there is no need to search and test all its 2-way interactions. All 2-way interactions are operational. Only interaction (a,b,e,f) is not covered. So the testing workloads of configuration (a,b,e,f) are reduced to test interaction (a,b,e,f) only to finalize the test result of configuration (a,b,e,f).

4.2.2 Algorithm Time Complexity Analysis

No matter F-table or P-table, the worst condition is that all stored results are traversed. Testing all configurations from 2-way to 6-way can cover almost all possible configurations. For n components, the number of all possible configurations is $C_n^2+C_n^3+C_n^4+C_n^5+C_n^6$ with time complexity O(n^6).

4.3 TA Analysis Process and Related Considerations

4.3.1 Analysis Process

1. 2-way TA analysis:

   (a) Use proposed search algorithms and distribute candidate configurations into different processors for execution. Different distribution methods can be used, such as based on the similarity among different configurations, the usage of different configurations, or the mixed methods;

   (b) Use P_2 (`P` table for 2-way interactions) to reduce testing effort. A P_2 configuration will not be `N`, and will not be X_2 (`X` table for 2-way interactions), but maybe X_3 (`X` table for 3-way interactions).

   (c) Complete testing all 2-way configurations (thus some `N` configurations may be around), and store all the results at P_2, F_2, N_2, and X_3 tables.

2. 3-way TA analysis using 2-way data:

(a) Eliminate those 3-ways configurations that have X_2, F_2, and N_2 2-way configurations. Those 3-ways interactions are sent to X_3, F_3, N_3 tables.

(b) Divide the P_2 configurations into different sets of configurations, and send to different processors for analyzing.

(c) Use P_2 (2-way `P` interaction) to reduce testing effort.

(d) Complete testing all feasible 3-way configurations.

3. 4-way TA analysis using 2-way and 3-way data:

   (a) Eliminate those 4-ways configurations that have X_2, F_2, and N_2 2-way configurations, and X_3, F_3, and N_3 3-way configurations. Those 4-ways interactions are sent to X_4, F_4, N_4 tables.

   (b) Divide the P_2 and P_3 configurations into different sets of configurations, and send to different processors for analyzing.

   (c) Use P_2 and P_3 (3-way `P` interaction) to reduce testing effort.

   (d) Complete testing all feasible 4-way configurations.

The above steps can be repeated for 5-way, and 6-way TA analysis.

4.3.2 Adjustment in Analyzing

All the possible configurations can be divided into different sets for different processors for analyzing. For simplicity, it is better to assign a set of related configurations to the same server. If `X` and `F` configurations are found, the related configurations can be easily eliminated in one server. There is no need to coordinate the results among different servers. Otherwise, the coordination cost is high. For example, set_1 is allocated to $processor_1$ for analysis. $processor_1$ will select configurations or tenant applications in set_1 for analyzing, If everything is great, i.e., all `P`, the test results are great, the analyzing process can stop. Otherwise, it requires further testing.

Lots of bugs exist in the candidate interactions. It is different to know the distributions of bugs. The bugs may not be evenly assign to different servers. Stop testing if the fault rate is high. The `F` test results can be used to deduce other `F` interactions by TA saving significant effort.

Multiple ways allocate configurations to processors, such as:

- **Tenant membership:** Configurations of one tenant application are assigned to one server as much as possible. It reduces the coordination costs of configuration test results among different servers.

- **Functionality information:** Tenant applications, that implement the same or similar functions, often share same and closely-related configurations. Clustering these tenant applications based on functionality also increases the test efficiency.

- **Random:** Randomly assign candidate configurations to different clusters. Load balancing may not be considered in assigning process and some configurations may be tested by multiple processors for redundant testing.

- `P/F` **configuration allocation:** Allocate `P/F` configurations to processors at local cache. Each processor can use the assigned `P/F` configurations to analyze candidate configurations.

4.4 Test Database Design

4.4.1 `X` *and* `F` *Table Design*

When one `X` or `F` interaction is saved in databases, test results of related configurations can be deduced. The saved test results are shared to all servers.

The test results tables correspond to the n-way interactions that have their own tables. If too many results are saved in one table, the table will be split for efficiency. The usage of each interaction decides the storage position of test result. The test results of frequently used interactions are saved in the top of data table.

4.4.2 `P` *Table Design*

Different from `X` and `F` interactions, the `P` test results can be reused to reduce test workloads and increase the testing efficiency, but they cannot eliminate the untested configurations from testing considerations. The P-table can follow the same design as X-table and F-table such as placing high priority items on the top, and priority can be adjusted dynamically. The usage ranking mechanism and data migration rules can also be used. The difference is that all `P` interactions must be saved in corresponding data table.

However, TA operation rules for `P` are different from operation rules for `X` or `F`. Specifically, the n-way ($n \geq 3$) configurations contain 2-way interactions. For any operational n-way ($n \geq 3$) configurations, all their sub-interactions (2-way interactions) must be operational.

Unlike X-table or F-table, P-table stores all `P` interactions without any omitting. So another difference is that relationships exist and can be traced among different n-way interaction table. For example, $(a, b, c) \succ (a, b)(b, c)$ or $(a, c)(b, c)$. If $(a, b, c) = \texttt{P}$, (a, b), (a, c), and (b, c) in 2-way interaction table have connections with (a, b, c) in 3-

way interaction table.

A (n+1)-way configuration contains n-way configurations. The existing test results of n-way configurations can be used to reduce the testing workloads of (n-1)-way configuration. In this paper, TA build 2-way P-table first, then 3-way, 4-way, 5-way, until 6-way.

Testing n-way configuration includes testing all its sub-configurations and itself. For candidate n-way configuration, TA searches existing test results from n-way to 2-way P-table. If n-way interaction of candidate configuration can be found in n-way P-table, stop searching and use the found operational result. Otherwise, search (n-1)-way P-table to find all existing test results of its sub-interactions. If all its (n-1)-way interactions are found in P-table, stop searching and only n-way interaction itself needs to be tested to finalize its test result. If not, only the non-found interactions need to be tested. For non-found interactions, repeat the previous procedures from (n-2)-way to 2-way P-table.

The following two examples show the search process. Suppose configuration (a,b,c,d,e) has never been tested before.

- Configuration (a,b,c,d,e) has five 4-way sub-interactions (a,b,c,d), (a,b,c,e), (a,b,d,e), (a,c,d,e), and (b,c,d,e). If these five interactions are found in P-table, only interaction (a,b,c,d,e) itself needs to be tested to finalize the test result.

- If four of its 4-way sub-interactions (a,b,c,d), (a,b,c,e), (a,b,d,e), and (a,c,d,e) are found in P-table, only interaction (b,c,d,e) needs to be tested. Repeat the same process in searching sub-interaction of interaction (b,c,d,e) in P-table. The similar process will be repeated until all its saved sub-interactions are found in P-table. Only non-found sub-interactions and interaction (a,b,c,d,e) itself need to be tested to finalize the test result.

4.4.3 `N` and `U` Table Design

In these two tables, test results of saved interactions may be changed by testing or TA analysis. Except adding new results, the deletion of existing test results often happens in N-table and U-table. Decreasing the data movement costs needs to consider in data table design. When one of `N` or `U` interactions changes its status, the previous saved status is deleted and empty space is left in test database. It is not good to move saved data forward to fill the empty spaces immediately. The system allows test database has a ceratin number of empty spaces. When empty spaces reaches the threshold,the system moves saved data forward to fill the empty space. There is a tradeoff between system efficiency and data movement costs for choosing reasonable threshold.

4.5 Experiment

The authors have performed experimentation using simulation data and data from published eScience software. The authors are developing a SaaS using the published software in an eScience website (myexperiment.org) with software contributed by scientists worldwide. Each software with its components in the myexperiment.org can be treated as a tenant application, a collection of software can be incorporated as a SaaS system.

4.5.1 Simulation

Numerous simulations have been performed, and this section provides one example with 25 components, and each component has two options. The total number of test configurations is 2^{50} (approximately $1.13 * 10^{15}$). The experiments are done for t-way configurations for $2 \leq t \leq 6$. All simulations are run on Intel Core 2 Quad CPU

Table 4.1: The Number of T-way Configurations from 2-way to 6-way

Range	**Size**
2-way configurations	1,200
3-way configurations	18,400
4-way configurations	202,400
5-way configurations	1,700,160
6-way configurations	11,334,400
Configurations from 2-way to 6-way	13,256,560

Table 4.2: The Initial Setting Ups of Infeasible Configurations

Range	**Initial Infeasible Configuration Size**	**Related Infeasible Configuration Size**
2-way configurations	10	10
3-way configurations	100	560
4-way configurations	1,000	15,520
5-way configurations	10,000	286,080
6-way configurations	100,000	3,280,400

2.40GHz machine. The numbers of t-way configurations for this example are listed in Table 4.1:

The following three tables list the initial infeasible (Table 4.2), faulty (Table 4.3), and irrelevant (Table 4.4) configurations. Table 4.2 and 4.3 also listed the related infeasible and faulty configuration after. For example, if (A, B) is infeasible, (A, B, C) and (A, B, D) are all infeasible. Other than infeasible, faulty, and irrelevant configurations, the rest of configurations are either operational or unknown.

The irrelevant configurations are stored in the N-table. The initial N-table contains:

Table 4.3: The Initial Setting Ups of Faulty Configurations

Range	**Initial Faulty Configuration Size**	**Related Faulty Configuration Size**
2-way configurations	25	25
3-way configurations	8	1,158
4-way configurations	0	25,652
5-way configurations	0	361,592
6-way configurations	1	3,640,561

Table 4.4: The Initial Setting Ups of Irrelevant Configurations

Range	**Size**
All 2-way configurations	20
All 3-way configurations	200
All 4-way configurations	2,000
All 5-way configurations	20,000
All 6-way configurations	200,000

TA is then used to identify those configurations that need to be tested by first eliminating those configurations that have been identified to be `X`, `F`, or `N`. Other than infeasible, faulty, and irrelevant configurations, the candidate configurations are operational or unknown. The following attempts change the ratio of initial operational and unknown configurations to find the relationship between initial P-table and TA efficiency.

Table 4.5, 4.6 show the input to the simulation with different percentages of configurations, specifically 5%, 10%, 20%, 30%, 40%, and 50% of configurations have status of `P` (operational) in Table 4.5, and Table 4.6 show the corresponding data for `U` (unknown) configurations from 5% to 50% of initial configurations are operational

Table 4.5: The Different Initial P-table Settings

Range	**Size**					
Percentage	**5%**	**10%**	**20%**	**30%**	**40%**	**50%**
2-way configurations	57	115	229	344	458	573
3-way configurations	824	1,648	3,296	4,945	6,593	8,241
4-way configurations	7,961	15,923	31,846	47,768	63,691	79,614
5-way configurations	51,624	103,249	206,498	309,746	412,995	516,244
6-way configurations	210,672	421,344	842,688	1,264,032	1,685,376	2,106,720

(P).

Table 4.7 shows the results of simulation, the data demonstrated that consistently the TA has eliminated 97.982% of configurations from testing consideration.

4.5.2 Parallel Computing for TA *Analysis*

The workloads with different initial settings of P-table from 5% to 50% of the candidate testing configurations are evenly assigned to different number of PoDs (Portal on Demand). It compares the results of one, four, eight, and sixteen PoDs. Figure 4.4 shows the simulation results. Horizontal axis is the percentage of the candidate testing configurations as the initial P-table settings. Vertical axis it the TA analysis time and the unit of time is hour. According to the increasing of initial P-table size, the TA analysis time decreases. Due to the four cores of machine, the

Table 4.6: The Different Initial U-table Settings

Range	**Size**					
Percentage	**5%**	**10%**	**20%**	**30%**	**40%**	**50%**
2-way configurations	1,088	1,030	916	801	687	572
3-way configurations	15,658	14,834	13,186	11,537	9,889	8,241
4-way configurations	151,267	143,305	127,382	111,460	95,539	79,614
5-way configurations	980,864	929,239	825,990	722,742	619,493	516,244
6-way configurations	4,002,767	3,792,095	3,370,751	2,949,407	2,528,063	2,106,719

result of four PoDs is the best in this simulation. The coordination time among different PoDs affects TA analysis efficiency. The more PoDs are used, the more coordination time is. So the results of eight and sixteen PoDs are worse than four PoDs. Comparing single PoD, assigning workloads to multiple PoDs is a good way to increase TA analysis efficiency.

4.5.3 Discussion

A small SaaS may contain thousand of components and may take a long time for testing. Assuming a small SaaS has only one hundred elements, the total number of combinations is 2^{100}. To complete testing of the combinatorial testing of these combinations will take 5 days for a PC. However, if the number of components increases to 120, it will take 20 processors 3 hours to complete. If the number components

Table 4.7: The Related and Reduced Configurations with Different Settings

	5%	**10%**	**20%**	**30%**	**40%**	**50%**
Involved Configurations (a)	$2.55*10^8$	$2.41*10^8$	$2.15*10^8$	$1.88*10^8$	$1.61*10^8$	$1.34*10^8$
Reduced Configurations (b)	$2.50*10^8$	$2.37*10^8$	$2.11*10^8$	$1.84*10^8$	$1.58*10^8$	$1.32*10^8$
Configurations to be tested (c)	$5.15*10^6$	$4.88*10^6$	$4.34*10^6$	$3.80*10^6$	$3.25*10^6$	$2.71*10^6$
Reduced Workloads Percentage (b/a)	97.982%	97.982%	97.982%	97.982%	97.982%	97.982%

become 150, 20-processors will take years to complete. Thus, eliminating the number of tests will be essential.

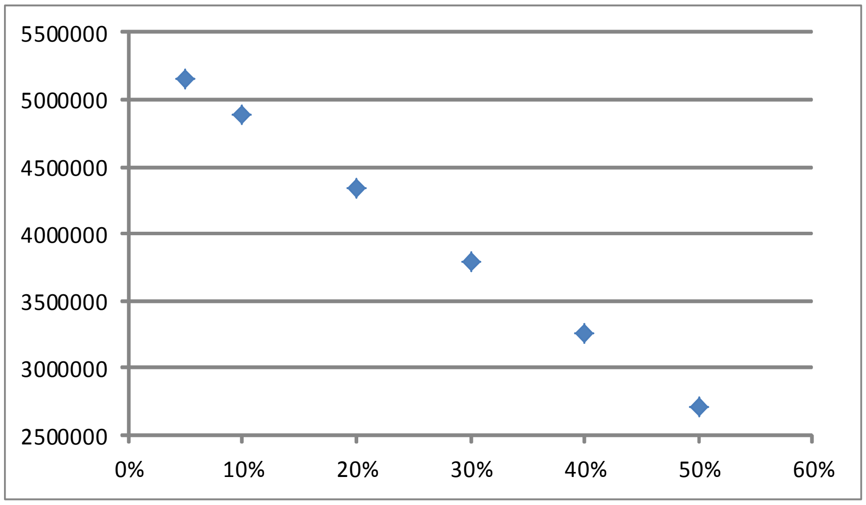
4.6 Conclusion

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

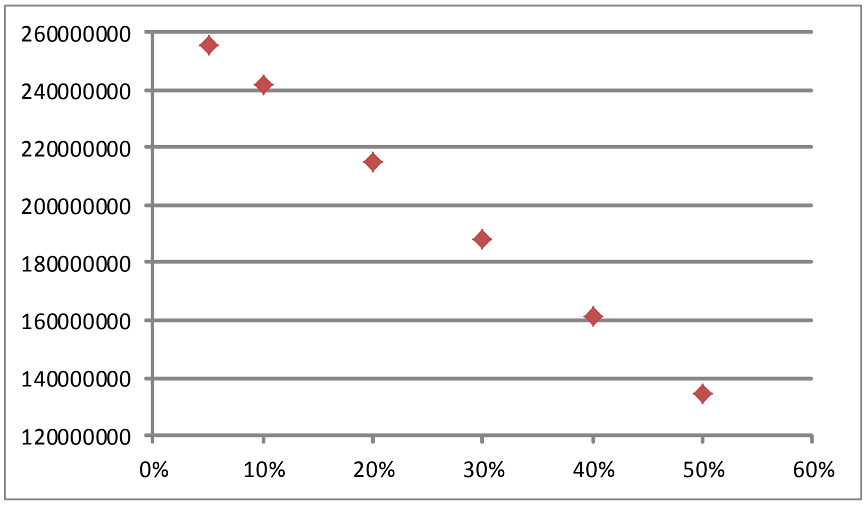
Table 4.8: Scalability Prediction of TA Analysis

Data Size	**Experiment Environment**	**Time**	**Feasibility**
2^{100}	Single PC	5 days	Feasible
2^{100}	20 machines (small cloud)	3 hours	Feasible
2^{120}	Single PC	14,364 years	Infeasible
2^{120}	20 machines (small cloud)	359 years	Infeasible
2^{150}	Single PC	$1.54 * 10^{13}$ years	Infeasible
2^{150}	20 machines (small cloud)	$3.86 * 10^{11}$ years	Infeasible
2^{200}	Single PC	$1.74 * 10^{28}$ years	Infeasible
2^{200}	20 machines (small cloud)	$4.34 * 10^{26}$ years	Infeasible

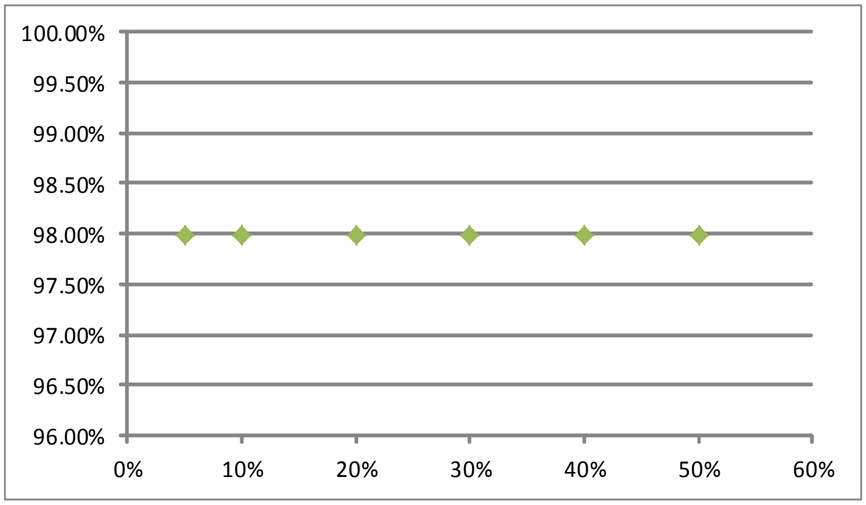
can be carried out in a cloud platform with a large number of processors to perform test execution in parallel to identify faulty interactions.



(a) Number of Candidate Configurations



(b) Number of Related Configurations



(c) Percentage of Reduced Workloads

Figure 4.3: The Simulation Results

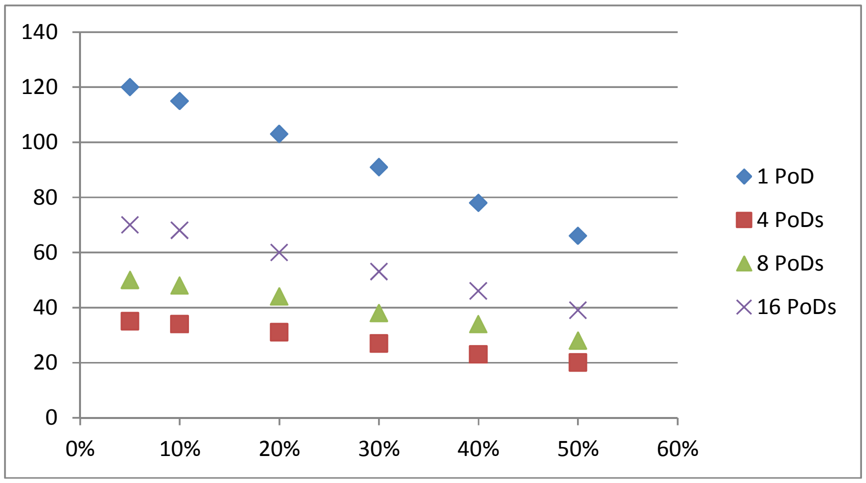


Figure 4.4: The Simulation Results of TA Time Efficiency

Chapter 5

TEST ALGEBRA EXECUTION IN A CLOUD ENVIRONMENT

5.1 TA Concurrent Execution and Analysis

5.1.1 TA *Concurrent Execution*

Figure 5.1 shows the relationship between TA and AR. The test database that contains X, F, P, N, U tables is shared to TA and AR. TA and AR can do concurrent execution on their test workloads.

As mentioned, AR needs multiple configurations to test to determine the status of interactions. Sometimes AR needs to test thousands of configurations before it can determine the faulty interactions. In AR it is the P that is useful as it can eliminate many candidates from testing. A *Pass* in AR will result in all of sub-configurations to pass. But in TA X and F are useful as it can eliminate, one X or F can eliminate many configurations. Thus, the strategy is to wait until sufficient

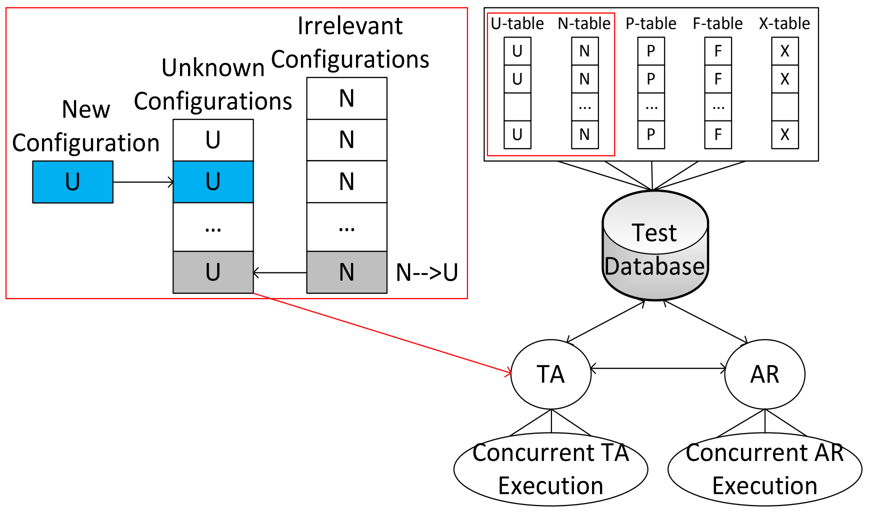


Figure 5.1: Relationship Between TA and AR

number of configurations (classified as `U`) to test, given a PTR (Previous Test Results). Run until any interactions have been identified as `X` or `F`, then run TA. Similarly, irrelevant configurations can also be eliminated from testing consideration. So only `U` configurations need to be tested.

New `U` configurations are put into candidate-configuration set. There are two types of new configurations:

1. Totally new configurations (it has not been tested, even though its sub-configurations may have been tested before.)

2. `N` configurations change to `U` configurations (also, some sub-configurations may have been tested before.)

Figure 5.2 shows that TA and AR share the same test database that includes `X`, `F`, `N`, `P`, and `U` tables. When new configurations come, they are added into `U` configuration set as candidate testing configurations. (type 1) Parts of `N` configurations change their statuses to `NU` (this is a new status and will be discussed soon) and are treated as `U` configurations. The `NU` and `U` configurations will be evaluated by TA and AR to identify faulty interactions using existing test results:

1. Run TA to check whether existing test results can be used to determine if the new configuration is valid. If yes, change status of new configurations from `U` to `X` or `F` only. It cannot be `P` as this is a new configuration and thus it must be tested.

2. Otherwise, wait for a sufficient number of configurations need to be tested to run AR to test new configurations. The explored test results are saved in test databases.

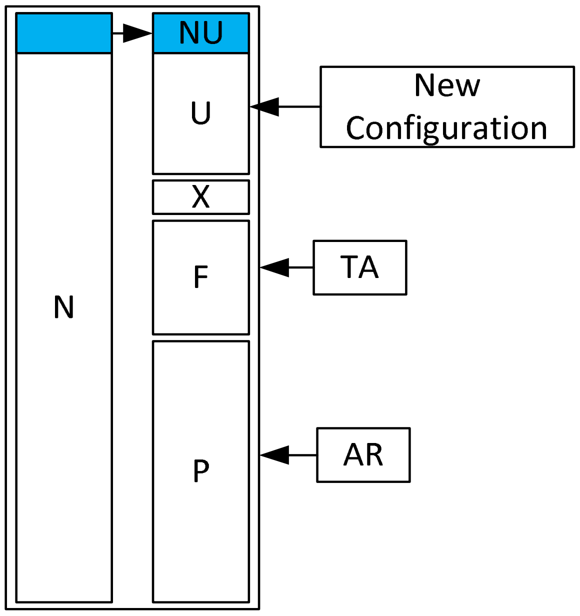
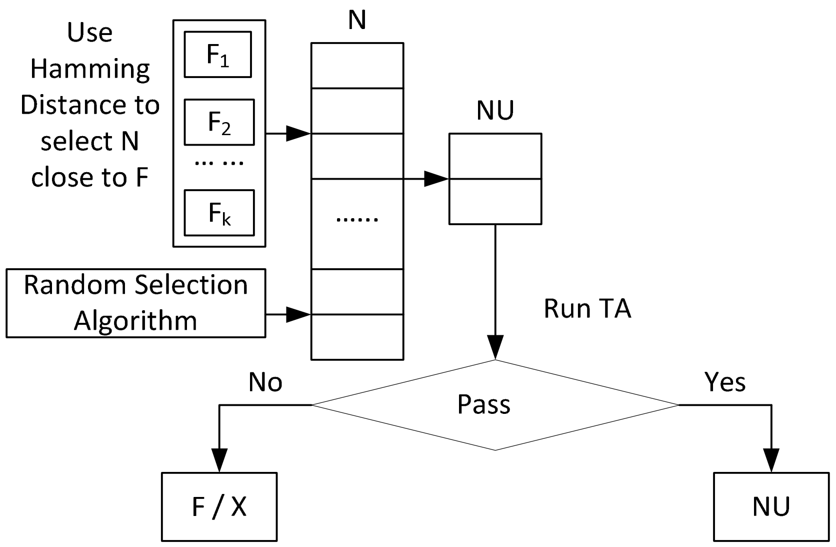


Figure 5.2: Test Results Shared by TA and AR

- If test result of new configuration is `F` or `X`, its faulty or infeasible interaction will be identified by AR.
- As most configurations will be `N`, even if AR runs all the `U` configurations, no faulty interactions have been identified. In this case, it is not productive to perform TA as TA needs `X` or `F` to eliminate configurations from testing. If so, we have two choices:
  - Stop testing (including both AR and TA) as no new information is available for further computation.
  - Or, convert some `N` configurations into `U`. These `N` configurations need not be tested, but they were tested to identify faulty interactions. They are labeled as `NU` (as they are actually `N`, but treated as `U`), and run AR with both `NU` and `U` configurations. Different algorithms can be developed to identify those `N` configurations to be re-labeled as `NU` so that these `NU` can be tested.


Use Hamming Distance to select N close to F
F1
F2
... ...
Fk
N
......
NU
Random Selection Algorithm
Run TA
No
Pass
Yes
F / X
NU
Figure 5.3: NU Configuration Selection Process

3. Change interaction that changes from N or U to F or X, if AR is successful in identifying them, run TA to eliminate all related configurations from testing consideration.

5.1.2 NU Configuration

As NU configurations are added into testing consideration, the number of N configurations and the number of U configurations change.

- Number of N configurations (N'): N' = N - NU
- Number of U configurations (U'): U' = U + NU

The total number of N' and U' equals to the total number of N and U. The only change is the number of NU in N and U configuration sets.

5.1.3 `NU` Configuration Selection Algorithms

Random Algorithm

Random algorithm shown in Algorithm I that select configurations randomly from `N` configuration set and change the selected configuration to `NU`. Random selection algorithm cannot involve `X`, `F`, `P`, and `U` sets.

Algorithm 4 Random Algorithm

```
Input:
    Irrelevant set of configurations
Output:
    Selected configurations
1: for int i = 0; i $\leqslant$ m; i++ do
2:     Randomly select one configuration out of irrelevant set
3:     Change the status of selected configuration from N to NU and return it
4: end for
```

Hamming Distance Algorithm

- **Definition:** The Hamming distance d(x, y) between two vectors $x, y \in F^{(n)}$ is the number of coefficient in which they differ where n is the number of components. For example:
  - $F^{(3)}((a, b, c), (a, b, d)) = 1$
  - $F^{(4)}((a, b, c, d), (e, f, c, d)) = 2$
- **Nearest Neighbour:** Given a code $C \in F^{(n)}$ and a vector $y \in F^{(n)}$, then $x \in C$ is a nearest neighbour to y if $d(x, y) = min(d(z, y)|z \in C)$. A vector

might have more than one nearest neighbour, so a nearest neighbour is not always unique.

Find those `N` configurations that have minimum Hamming distance between existing `F` configurations, and then change their statuses to `NU`. The actual conditions may have different minimum Hamming distance. Using minimum Hamming distance to find those `N` configurations closely related to `F` configurations increases possibility to find those potential faulty configurations in `N` set.

It uses two examples to show how to use Hamming distance to select `NU` configurations from `N` configurations.

- It sets one as the default minimum Hamming distance. Select all configurations in `N` set that have one Hamming distance between selected faulty configuration. Change the statues of these selected configurations from `N` to `NU`. Suppose 3-way interaction (a, b, c) is faulty. (a, b, d) is one Hamming distance away between interaction (a, b, c), so it can be selected as `NU`.

- Figure 5.4 shows the process of finding all mutations from `F` interaction. Suppose 2-way interaction (a_1, b_1) is faulty. 4-way interaction (a_1, b_1, c_1, d_1) is also faulty. Each component in 4-way interaction has three options. One can have the following.

  - *Mutation in faulty interaction:* (a_2, b_1, c_1, d_1), (a_3, b_1, c_1, d_1), (a_1, b_2, c_1, d_1), (a_1, b_3, c_1, d_1), ... Only faulty combination part of 4-way interaction mutates and the remaining part keeps same. The minimum Hamming distance is one.

  - *Mutation in both faulty and non-faulty interaction:* (a_2, b_1, c_2, d_1), (a_3, b_1, c_2, d_1), (a_1, b_2, c_1, d_2), (a_1, b_3, c_1, d_2), ... The minimum Hamming distance is two.

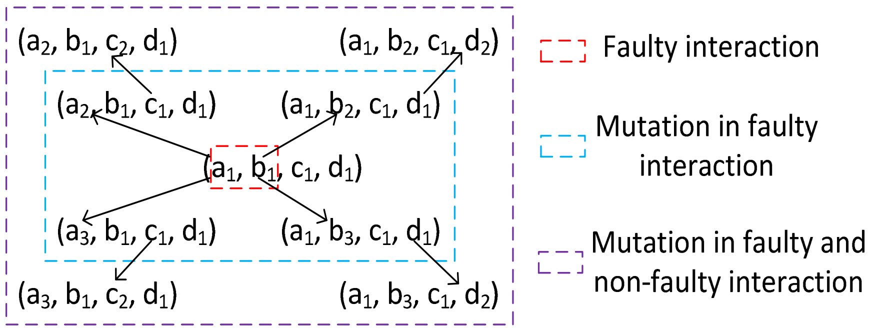


Figure 5.4: Using Hamming Distance to Select NU Configuration

Mixed Strategy

The two algorithms proposed earlier can be used together. In other words, some random configurations will be used together with Hamming distance algorithm from faulty interactions with various Hamming distances, say from one to three. As often combinatorial testing has low failure rate, using the random algorithm will result in status of P often, and thus speed up the AR algorithm. If the Hamming distance algorithms can detect some X or F interactions, it will help TA to eliminate configurations from testing.

5.1.4 Analysis Process of NU and U Configurations

NU configuration selection process shown in Figure 5.3:

1. Use random algorithm or Hamming-distance method to select candidate configurations from N configurations into the initial set of NU configuration.

2. Use TA to analyze the selected NU configurations.

3. If they do not pass TA analysis, they must be X or F, and they will be removed from the set of NU configurations.

4. Those that passed TA analysis can be used by AR for testing.

5. After AR testing, perform TA on those configurations that are in U or U + NU. TA can be triggered by the following two ways:

   - *On-demand:* run TA whenever AR detects a F or X.
   - *Batch:* run TA when AR detect a certain number of F or X, or when both U and NU have been tested completely, whatever criteria gets fulfilled. The number can be determined experimentally.

As even just one new F or X is detected by AR, numerous configurations can be removed from testing by TA, thus even with the batch mode, the number of X or F detected need not be large.

In the integrated process, AR and TA analysis are activated by F configurations. F configurations are identified by testing. When U configurations are tested and all existing F configurations are analyzed, AR and TA analysis stop. No more related F configurations can be identified by testing analysis. For eliminating more configurations from testing considerations, it needs more F configurations to explore those N configurations.

Some N configurations that are closely related to existing F configurations can be converted into U, marked as NU, and treated as U. Test results of these NU configurations can be finalized by testing. The identified faulty NU configurations are analyzed by AR to identify the faulty root. Once the faulty root is identified, TA is activated to analyze those U configurations and eliminate related F configurations. The details of NU configurations analysis are shown in NU configurations processing algorithm.

5.1.5 On-demand Interaction Testing with PTR

The test result of interaction (a, b) is not known, i.e., U or NU. One configuration is $(c_1, c_2, ..., c_k)$. Suppose a and b are c_1 and c_2 respectively. So choose c_3,... to c_k

Algorithm 5 NU Configurations Processing Algorithm

```
Input:
    F, N, P, U configurations
Output:
    deduced F configurations, updated U configurations
1:  Run Hamming distance algorithm to find N configurations that are closely related
    to F configurations
2:  Mark the found N configurations as NU
3:  Run test cases on NU configurations
4:  if any NU configuration is faulty then
5:      Run AR
6:      Return identified F interaction
7:  end if
8:  while all related F configurations are eliminated based on existing test results
    do
9:      if F interaction exists && U configuration exists then
10:         Run TA
11:         Return identified F configurations & updated U configurations
12:     end if
13: end while
```

from a `P` configuration in the PTR, make it cc_1 (candidate configuration 1). Do that cc_2, cc_3, ... and so on. Following this, choose as many configurations from the set of `P` configuration. On-demand interaction testing process:

1. Run cc_1, if it passes, interaction (a, b) pass. There is no need to run cc_2, cc_3,... and so on. If cc_1 fails, there are several possibilities:

   (a) interaction (a, b) fails

   (b) Or interaction (a, b, xxx) fails, for 3-way to 6-way, for example, 6-way will be (a, b, c, d, e, f)

   (c) Or interaction $(a, yyyy)$ fails, from 2-way to 6-way, for example, 2-way will be (a, c)

   (d) Or interaction $(b, zzzz)$ fails, from 2-way to 6-way, for example, 2-way will be (b, c)

   The above possibilities needs to be distinguished.

2. Run cc_2, if it passes, interaction (a, b) passes, so choices will be the above b), c), or d) only. If cc_2 fails, it has similar cases as the above 4 possibilities. As cc_1, and cc_2 are different, if cc_1, and cc_2 share interaction (a, b) only, then it can conclude that interaction (a, b) fails now. Formally, common $(cc_1, cc_2) = (a, b)$, then $V(a, b) = \texttt{F}$. Stop as it can conclude (a, b) is `F`. Otherwise, it continues.

3. Run cc_3, if it passes, interaction (a, b) is `P`. If cc_3 also fails, it has similar cases as case 2).

One can now run cc_1, cc_2, cc_3,... to cc_k in this manner. And k can be predetermined. As each cc_i has same pf (probability of failure) and this is low, k should be small. Furthermore, can run this sequentially. For example, pf = 0.1%, this means, 99.9% of time, cc_1 will pass. If not, 99.9% of time, cc_2 will pass, assuming that they are independent.

It has $(1 - pf)^m$ where m is the number of cc_i that it tries to test for each interaction (a, b). For 2-way (a, b), pf is low. For 3-way (a, b, c), pf is even lower. For 4-way (a, b, c, d), pf will be even lower, thus, m and k need be small.

Operation procedure (common(exp1, exp2) = common elements in both exp1 and exp2)

1. Test cc_1, if pass, return $V(a, b) =$ `P` and stop (with probability of $1 - pf$).

2. Otherwise test cc_2, if pass, return $V(a, b) =$ `P` and stop (with probability of $pf * (1 - pf)$). If common(cc_1, cc_2) $= (a, b)$, stop and return $V(a, b) =$ `F`. Otherwise it proceeds to test cc_3.

3. Test cc_3, if pass, return $V(a, b) =$ `P` and stop (with probability of $pf^2 * (1 - pf)$). If common(cc_2, cc_3) $= (a, b)$ or common (cc_1, cc_3) $= (a, b)$, return `F`, and stop.

4. Otherwise test cc_4, if pass, return $V(a, b) =$ `P` and stop (with probability of $pf^3 * (1 - pf)$). If common(cc_1, cc_4) $= (a, b)$ or common(cc_2, cc_4) $= (a, b)$ or common(cc_3, cc_4) $= (a, b)$, return `F`, and stop.

5. It runs the the same procedure to test cc_5 to whatever cc_k, until the m reaches k. Or it can continue until k is 10 or 20. Most likely k should be less than 6.

For this reason, it is best to choose cc_i as separate as possible, for example, assuming each configuration has m elements, it likes cc_i to share hopefully 2 common elements only (assuming for 2-way interaction, for 3-way interaction, it likes to have hopefully 3 common elements only, and so on). If after say cc_6, all of them return `F` from testing, we have pf^6, very rare event that is very unlikely to happen. Assuming pf =

1%, that means 0.01^6. So, it has an optimistic algorithm that is likely to succeed to perform on-demand testing.

The algorithm is adjustable, depending on pf value, one can increase m (or k). For example, the larger pf, more testing will be needed. If one wants to have higher confidence, increase m (or k), for example, put m to 30, in that case, the worst case scenario will be pf^{30}, and yet it still cannot conclude if interaction (a, b) causes the failure. Furthermore, construction of cc_i from PTR can be random too. So, common(cc_i,cc_j) may overlap more than (a, b), but as m is large, common(cc_i,cc_j) $=$ (a, b) will eventually increase. If random selection, it can put m or k to be higher.

Figure 5.5 shows five trials with different pf 0.001, 0.00075, 0.0005, 0.00025, 0.0001. The probability of reaching 30 failures in on-demand interaction testing is simulated. The horizontal axis is the value of pf. The vertical axis is the simulated results in logarithm value. The simulation process follows the formula $iteration\ result = pf^{n-1} * (1 - pf)$, $n \in Z$ and $n \geq 1$. After thirty times running, no matter what initial pf is, iteration value approaches zero.

5.2 TA Experiments

5.2.1 TA MapReduce Experiment Flow Chart

Figure 5.6 shows how the TA MapReduce experiment goes. The whole experiment executes on-demand TA analysis. Input and output of TA MapReduce experiment:

- *Input:* Configurations and seed faulty interactions
- *Output:* TA efficiency and running time

Experiment environment:

- *Cluster:* 50 nodes Hadoop cluster(each node has 8 processors)

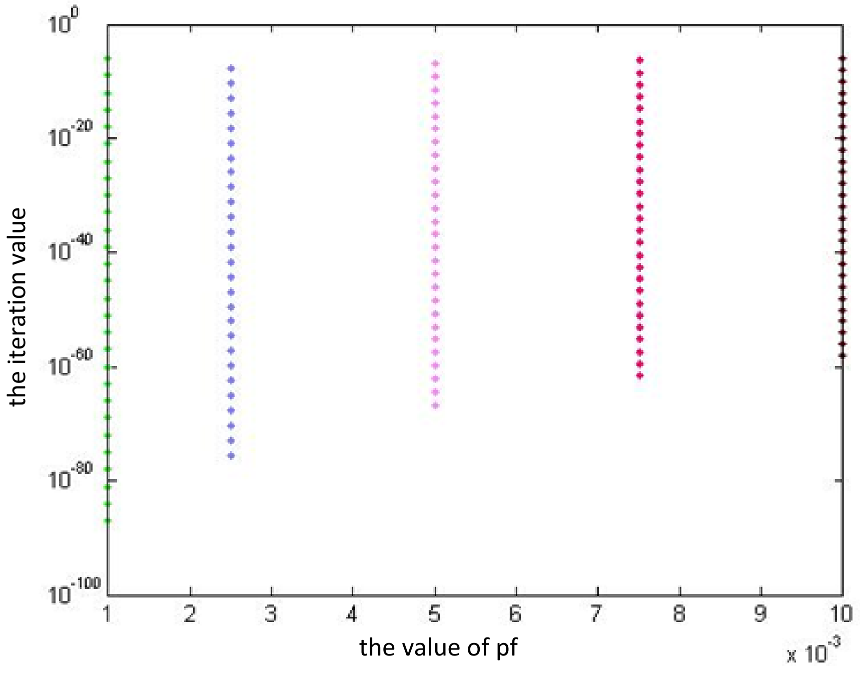


Figure 5.5: Five Trials with Different PF

- *CPU Processor:* Intel Xeon CPUE5520 2.27GHz
- *Memory:* 11G
- *Operating System:* CentOS release 6.3 (Final)
- *Hadoop Version:* 1.1.2
- *HBase Version:* 0.94.12

5.2.2 *Different Configuration Numbers of* TA *Experiments*

The first experiment is the effect of configuration number on TA efficiency. This experiment proceeds without any speedup strategy. Figure 5.7 shows that TA has a good performance on reducing test workloads. TA efficiency can also be improved a little when the number of configurations grows. The default fault rate is 0.001 in this experiment.

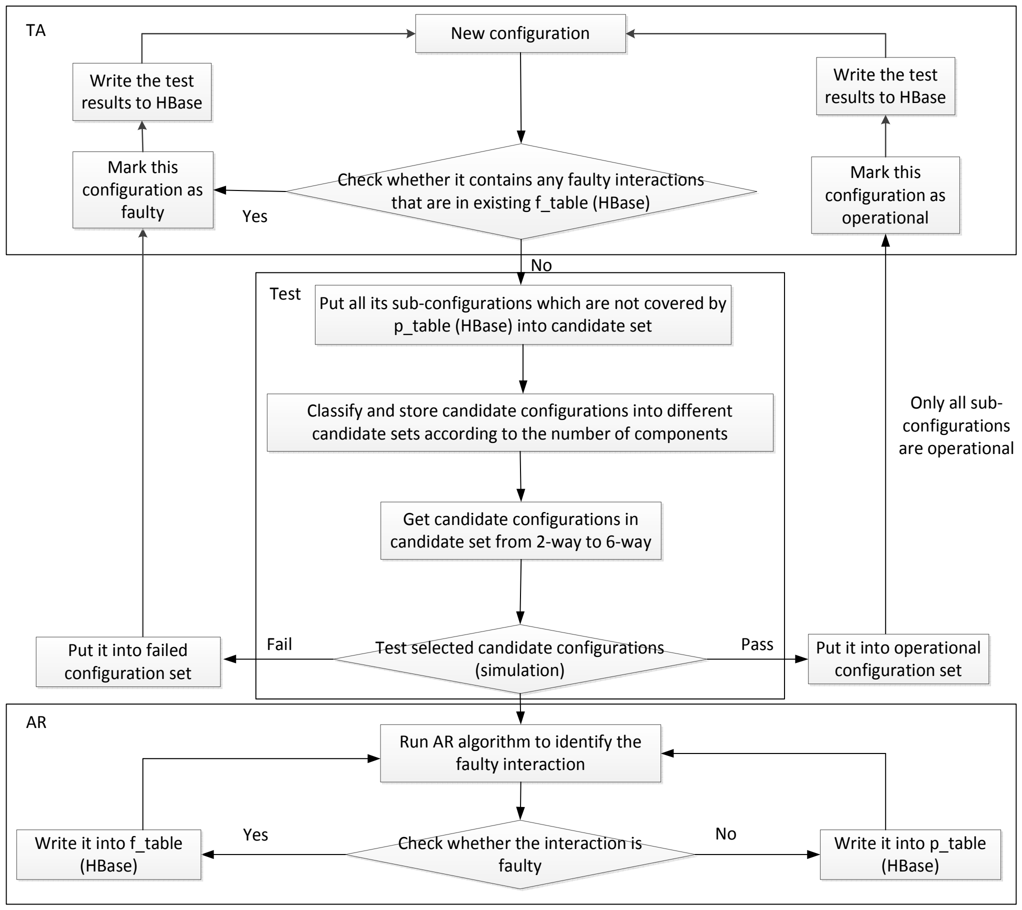

TA
New configuration
Write the test results to HBase
Write the test results to HBase
Mark this configuration as faulty
Check whether it contains any faulty interactions that are in existing f_table (HBase)
Yes
Mark this configuration as operational
No
Test
Put all its sub-configurations which are not covered by p_table (HBase) into candidate set
Classify and store candidate configurations into different candidate sets according to the number of components
Only all sub-configurations are operational
Get candidate configurations in candidate set from 2-way to 6-way
Fail
Test selected candidate configurations (simulation)
Pass
Put it into failed configuration set
Put it into operational configuration set
AR
Run AR algorithm to identify the faulty interaction
Yes
No
Write it into f_table (HBase)
Check whether the interaction is faulty
Write it into p_table (HBase)

Figure 5.6: TA MapReduce Experiment Flow Chart

5.2.3 Different Speedup Strategy for TA *Experiments*

This experiment speeds up TA process with four different strategies (fault rate: 0.001).

- *No Strategy:* Do not use any speedup strategy
- *Bloom Filter:* Use hash-map method to store the information of interactions (Bit Storage)
- *Table Splitting:* Split `F` and `P` table into five tables respectively according to way number of interactions (from 2-way to 6-way)

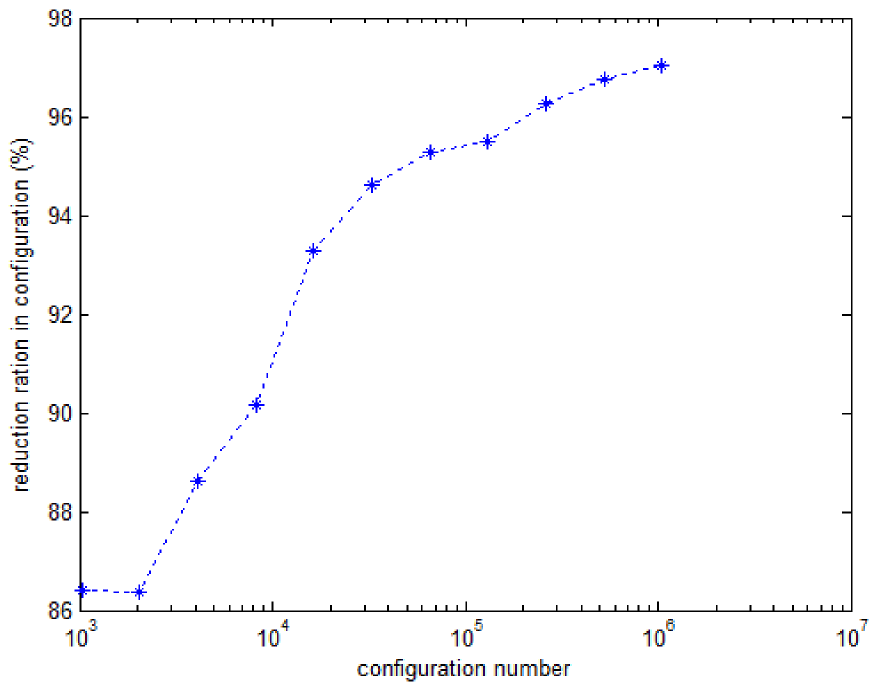


Figure 5.7: TA Efficiency on Hadoop

- *Mixed Strategy:* Use Bloom Filter and Table Splitting

Figure 5.8 compares the running time with different strategies. Bloom Filter does not affect much on running time, while Table Splitting speeds up test process significantly. Figure 5.9 shows the effect of different speedup strategies on TA efficiency. TA efficiency varies a little bit when configuration number is small. When configuration number increases, all strategies will have the same efficiency.

5.2.4 Different Fault Rates for TA *Experiments*

This experiment explores the effect of different fault rates on TA efficiency (configuration number: 524228). Figure 5.10 shows TA efficiency is affected by different fault rates. TA efficiency improves when the fault rate grows, as the fault rate can enhance `F`-table checking process. Figure 5.11 shows the effect of different fault rate on running time. Fault rate can decrease the running time, for the same reason.

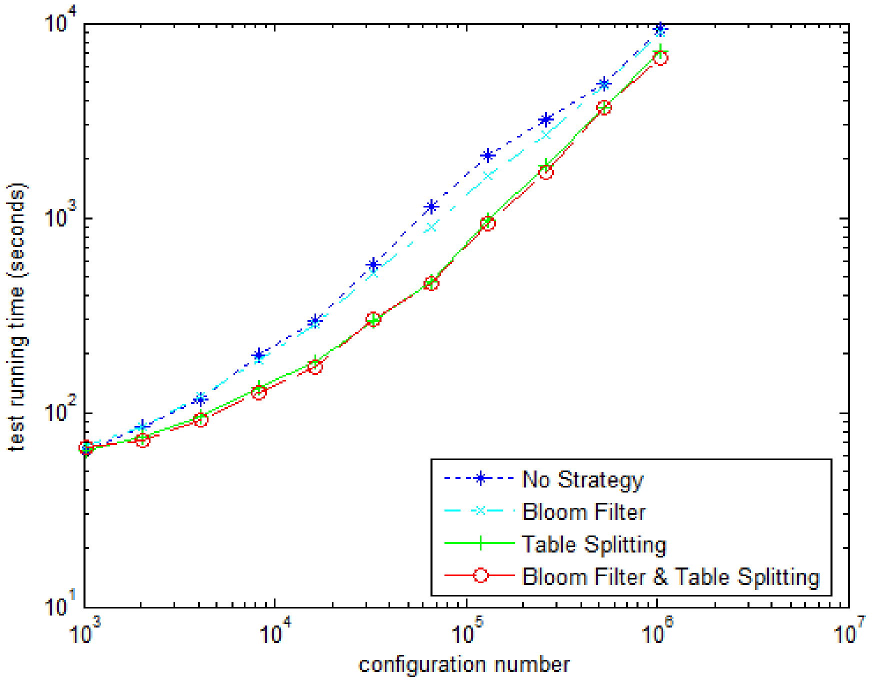


Figure 5.8: Running Time with Different Strategies on Hadoop

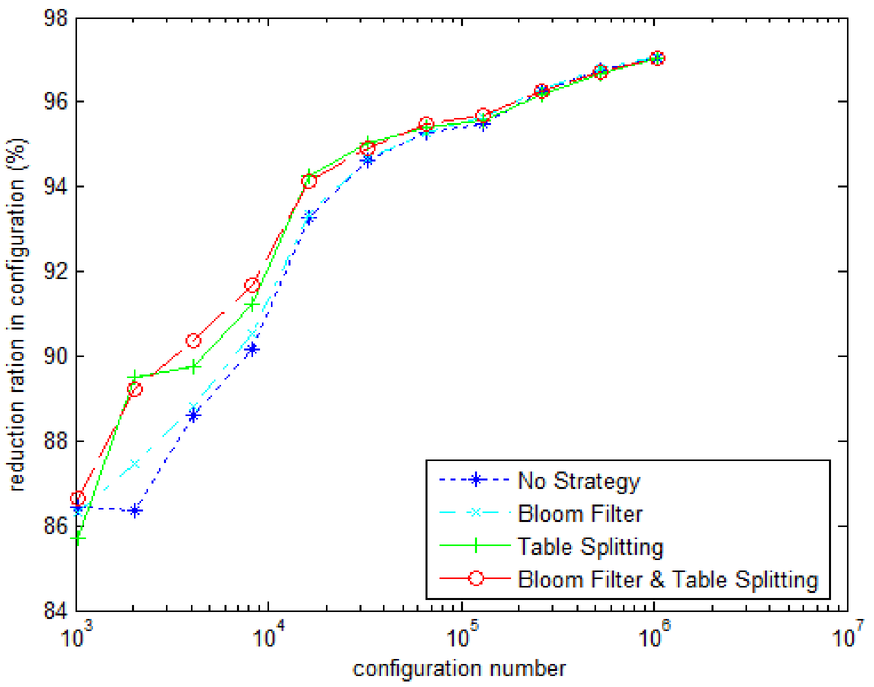


Figure 5.9: TA Efficiency with Different Strategies on Hadoop

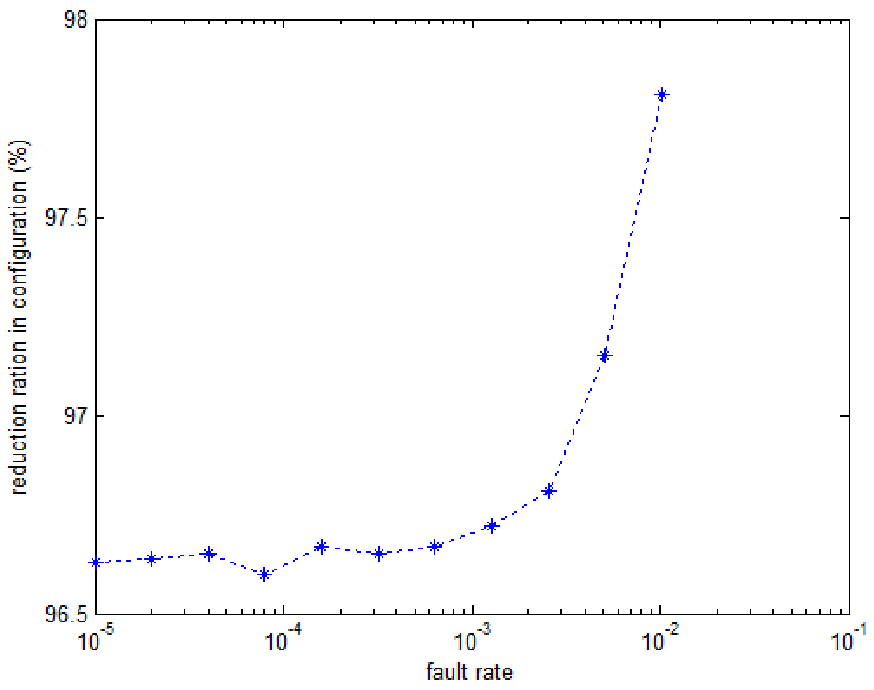


Figure 5.10: TA Efficiency with Different Fault Rates on Hadoop

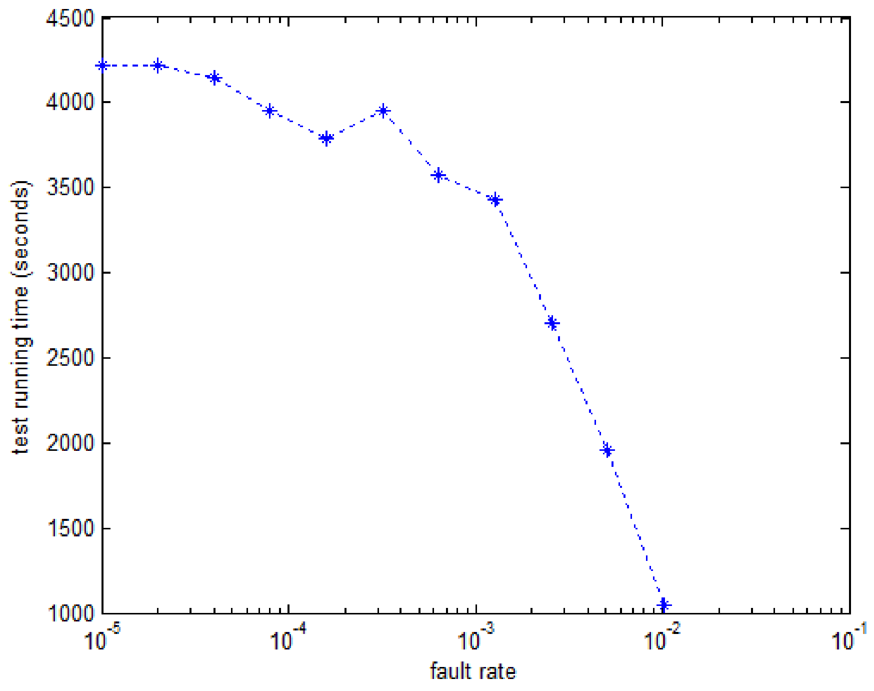


Figure 5.11: Running Time with Different Fault Rates on Hadoop

Table 5.1: Explanations of Each Parameter in Simulation

Parameter	**Meaning**	**Default Value**
$COMPONENTS\_NUMBER$	The number of components	1000
$VALUES\_NUMBER$	The values number for each component	2
$GUI\_PERCENT$	The percentage of GUI components	40%
$WORKFLOW\_PERCENT$	The percentage of workflow components	30%
$SERVICE\_PERCENT$	The percentage of service components	20%
$DATAMODEL\_PERCENT$	The percentage of data model components	10%
$TENANT\_APPLICATIONS_NUMBER$	The number of tenant applications	512
$COMPONENTS\_NUMBER\_IN\_APP$	The components number in one application	10
$ERROR\_PROBABILITY$	The fault rate	0.001

5.2.5 Explanation on Simulated Data

Table 7.1 shows the parameters used. The total number of configurations equals to the number of tenant applications powered by number of values and number of components in applications.

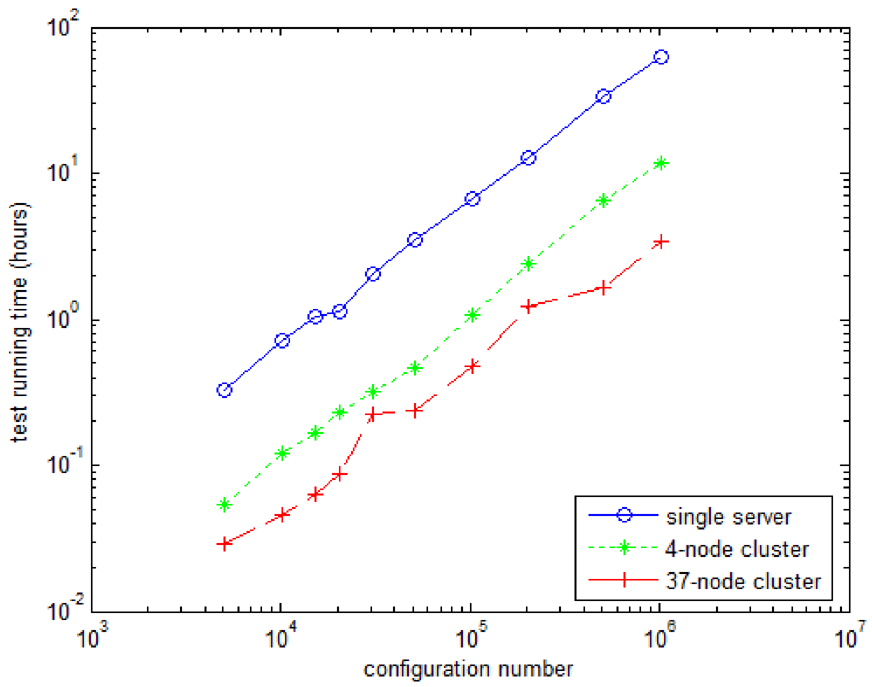


Figure 5.12: Running Time of TA Implementation on Hadoop Using Different Clusters

5.2.6 Simulation with Different Clusters

Figure 5.12 and 5.13 show the TA efficiency with different number of configurations. The x-axis represents the number of configurations, and the y-axis presents the TA efficiency. 97% of test cases can be reduced by TA algorithm. The improvement depends upon the fault pattern and TA algorithm. Parallel implementation of TA algorithm has no contribution to the TA efficiency, although it can greatly shorten the execution time of TA processing.

5.2.7 Simulation using 37-node Cluster with Differen Map Slots

The experiments were done using 37-node cluster. Figure 5.14 and 5.15 show the TA efficiency with different map slots on each machine. The x-axis represents the number of configurations, and the y-axis presents the TA efficiency. The number of map slots represents the number of map tasks each machine can process at same

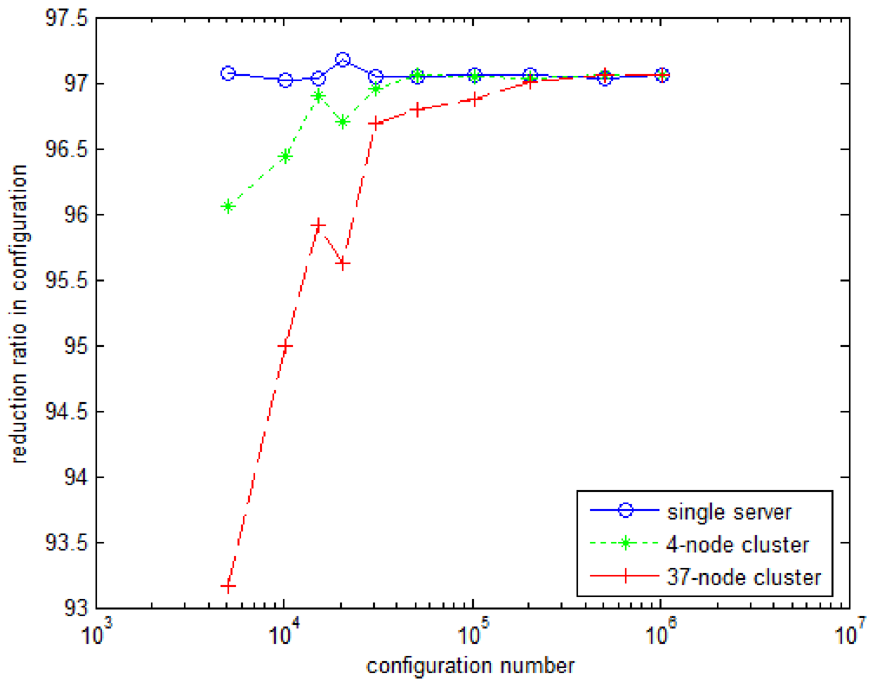


Figure 5.13: Configuration Reduction Ratio Using TA Implementation on Hadoop Using Different Clusters

time. Generally, more map slots are used, the less execution time is. But, more map slots actually contributes almost nothing to the execution time. It is probably as the limitation of HBase ability to handle the requests. Also, more map slots has nothing to do with the TA reduction ratio as Figure 5.15 shows.

5.3 Conclusion

The TA defines five states of test results with a priority and three operations, provides a foundation for concurrent combinatorial testing. 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 failure has been detected so that retest is needed. TA and AR cooperates to ana-

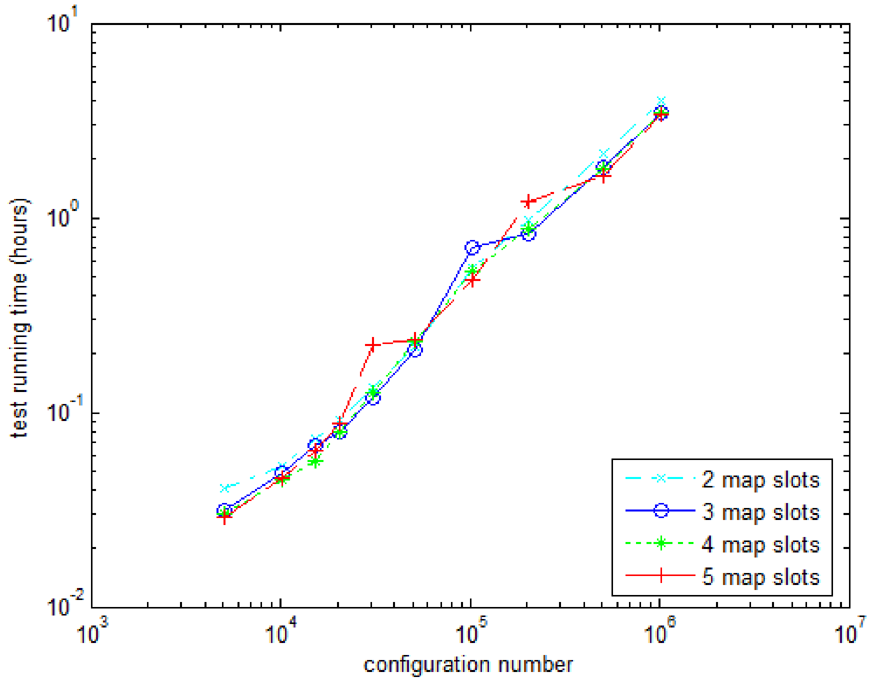


Figure 5.14: Running Time of TA Implementation on Hadoop Using 37-node Cluster with Different Map Slots

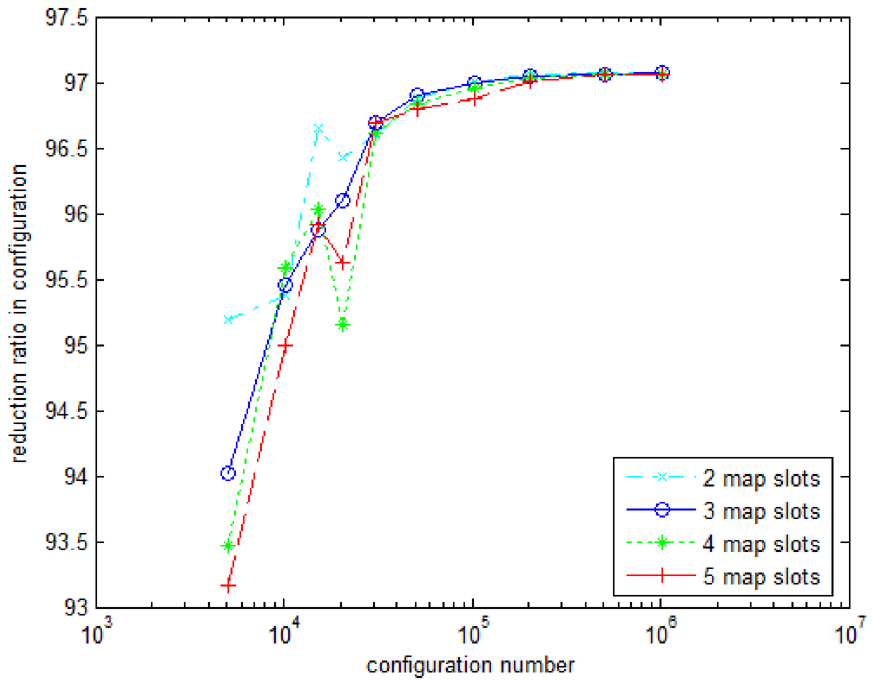


Figure 5.15: Configuration Reduction Ratio Using TA Implementation on Hadoop Using 37-node Cluster with Different Map Slots

lyze candidate configurations for increasing testing efficiency. 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.

Chapter 6

TAAS (TESTING-AS-A-SERVICE) DESIGN FOR COMBINATORIAL TESTING

6.1 TaaS Introduction

6.1.1 TaaS Definition

Several TaaS definitions are available Gao *et al.* (2011a); Riungu *et al.* (2010). It often means that testing will be online, composable, Web-based, on demand, scalable, running in a virtualized and secure cloud environment with virtually unlimited computing, storage and networking. This paper proposes a TaaS definition from two perspectives: **user's point of view** and cloud **internal point of view**.

From **user's point of view**, TaaS provides the following four services.

Test Case and Script Development: Users can develop, debug, and evaluate test cases/script online using automated tools in a collaborative manner. Test scripts may even be developed by customizing/composing existing components following the MTA approach.

Test Script Compilation and Deployment: Test scripts can be compiled and deployed for execution in a cloud environment, and TaaS resource management can allocate and reclaim resources to meet the changing workload.

Test Script Execution: Test can be executed in parallel or in a distributed manner, and it can be triggered autonomously or on demand.

Test Result Evaluation: Cloud-based test database is built to support automated data saving, intelligent retrieval, concurrent transaction, parallel processing, and timely analysis of huge test results.

From cloud **internal point of view**, TaaS may have the following features common to most cloud operations.

Decentralized Operations: Testing tasks may be executed in a parallel or a distributed manner, migrated to dynamic allocated resources, and performed in a redundant manner, or embedded within other cloud operations.

Metadata-based Computing: Controller uses metadata to control test operations such as time, frequency, multi-tasking, redundancy, parallel execution. TaaS metadata may include information about test scripts, cases, environment, and results such as index, location, and organization.

Data-centric Testing: Big Test handles large sets of input data and produces large sets of test results. Techniques for Big Data storage, processing, and understanding are key to TaaS. For examples, test data can be saved in in-memory databases, classified by attributes (such as hot, warm, or cold), and analyzed in real-time.

Multi-tenancy Test Script Composition: Like tenant applications in a MTA SaaS platform, test scripts in a TaaS system may share the same test script base.

Automated Test Redundancy Management and Recovery: Testing tasks can be partitioned and sent to different processors for parallel and redundant processing. Test and test results can be recovered in case of failures in a processor or in a cluster due to automated redundancy management. Recovery can follow the metadata-based approach.

Automated Test Scalability: When the SUT (System Under Test) scales up at runtime in a cloud environment, TaaS also needs to scale up proportionally using common cloud scalability mechanisms such as 2-level scalability architecture and s-tateless service design Tsai *et al.* (2012).

6.1.2 Three Generations of TaaS

First Generation of TaaS: In this TaaS generation, conventional testing or evaluation tools can be deployed to a cloud environment to provide an on-demand services with scalable resources. The cloud infrastructure such as PaaS or IaaS will manage resources by scheduling or deployment according to the testing workload dynamically. The testing software may take advantages of the resources and services provided by a cloud environment such as automated triplicate storage and computation as each task in a cloud environment such as GAE is computed three times and stored in different location for reliability. In this generation, a testing service is an application running on top of a cloud environment, thus it does not control the internal cloud scheduling or resource management. The testing software maybe the same or a slightly modified version of the conventional testing software.

Second Generation of TaaS: In this generation, testing software is an integrated part of a cloud environment by being implemented as a SaaS running on top of a PaaS. But a SaaS may have either full or limited control of cloud scheduling and resource management. As a SaaS may be fully integrated with a PaaS, and thus it may have an integrated scheduling and resource management capabilities, or a SaaS may be just an application program running on top of an existing PaaS without any control of scheduling or resource management in the PaaS.

A SaaS often provides customization, multi-tenancy (multiple software programs share the same code base), and scalability, and thus a TaaS implemented as a SaaS allows different testing services to be customized by different testing tenants, and multiple testing services can share the same testing code, and testing services can be scaled dynamically as controlled by TaaS.

If a TaaS is a SaaS fully integrated with PaaS, it will have significant control in scheduling and resource management, and the TaaS can make intelligent decisions as it will be able to assess the current cloud status.

If a TaaS is a SaaS application running on top of an existing PaaS, the TaaS customization, multi-tenancy, scalability can still be done, except that as TaaS does not manage resources directly, it needs to call the underlying PaaS to allocate resources, but other applications may be running at the same time to compete for the shared resources.

Third Generation of TaaS: In this generation, a TaaS is fully integrated into SaaS and PaaS where testing-related activities are performed autonomously and intelligently. For example, in a given SaaS, each tenant components will be tested by associated testing software automatically whenever they are checked into the SaaS database, and each time a tenant application is composed, the application is automatically tested by the associated testing software.

TaaS may also be fully integrated with policy management in a cloud environment where policies are enforced and evaluated at runtime to ensure various properties are held during execution.

A TaaS is also fully integrated with various monitoring capabilities of SaaS and PaaS, and a TaaS task may be composed, deployed, executed, scaled, and migrated like a regular SaaS or PaaS task.

6.2 TaaS Design with TA and AR

This section presents a new TaaS design for combinatorial testing using TA and AR. Figure 6.1 shows a TaaS design with six parts. There are SaaS components DB, Test Processing, AR, TA, Test Database, and Recommendation system.

Part I SaaS Components DB: Each tenant application in SaaS has components from four layers: GUIs, workflows, services, and data.

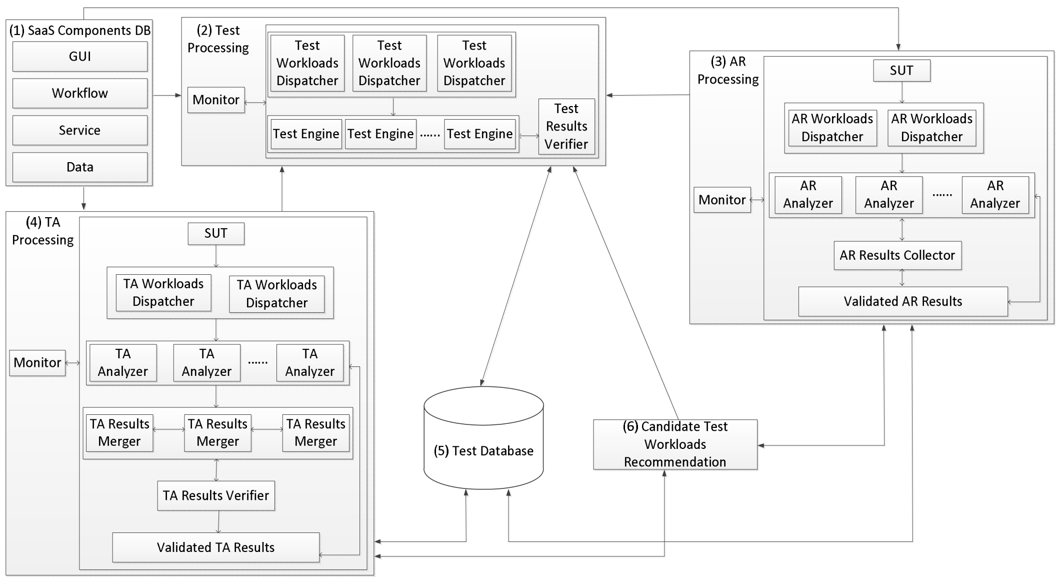


Figure 6.1: Taas Design for Combinatorial Testing Using TA and AR

Part II Test Processing: It uses the following components to process SaaS combinatorial testing.

Test Workloads Dispatcher: All testing workloads are sent to test dispatchers. Test dispatchers assign workloads to Test Engines according to the computation capacity of each Test Engine. The same workloads may be executed on different Test Engines for redundant testing.

- Input: candidate configurations, the number of Test Engines, the computation capacity of each engine; and
- Output: the amount of candidate configurations assigned to each Test Engine.

Test Engine: It runs different test cases to test the assigned workloads. Test results are sent to Test Results Verifier.

- Input: the assigned candidate configurations, test cases; and
- Output: test results of the assigned candidate configurations.

Test Results Verifier: It verifies all returned test results. For the same configuration, it may have different returned test results from different Test Engines. Test Result Verifiers finalizes the correct test result based on the confidence of each test result. Only those highly confident test results are saved in the Test Database and can be shared with others. If test results verifier cannot verify the returned test results, it requires Test Engines to retest these configurations.

- Input: test results from different Test Engines; and
- Output: finalized test results.

Monitor: It monitors the testing process. Test Workloads Dispatcher, Testing Workloads, and their related Test Engines are monitored. Each Test Engine is monitored during the testing process. Test Results Verifier is also monitored.

Part III AR Processing: It is used to figure out faulty configurations from the candidate set rapidly based on the existing test results.

SUT: It is the candidate test set.

AR *Workloads Dispatcher:* It works similarly as the Test Workloads Dispatcher of Test Processing. Different amount of candidate testing workloads are assigned to different AR Analyzers based on the computation capacity.

AR *Analyzer:* It runs AR algorithm on candidate configurations based on the existing test results. The analyzed test results are sent to the collector. It also reanalyzes those returned incorrect test results that did not pass validation.

- Input: existing test results, candidate configurations; and
- Output: test results of assigned candidate configurations.

AR *Results Collector:* It collects all test results from different AR Analyzers.

Collector also sends those candidate configurations that cannot pass test results validation to their related AR analyzers.

Validated AR *Results:* They save all validated AR results and send them to the shared test database. The saved validated results are shared to all AR analyzers.

Monitor: It is similar as monitor of Test Processing. The process of AR Analysis is monitored.

Part IV TA **Processing:** It analyzes test results by TA. Similar to AR Processing, TA also has SUT, test dispatcher, and monitor. Their functions are same as the corresponding parts in AR. The other parts of TA have their own features.

TA *Analyzer:* It runs TA to analyze the test results of candidate test set based on the existing test results. Test results of those candidate configurations related to existing X or F interactions can be finalized.

- Input: existing test results, candidate configurations; and
- Output: test results of candidate configurations, candidate interactions.

TA *Results Merger:* It merges the returned from different TA analyzers by three defined operations. The merged test results are sent to test result verifier.

- Input: test results from different analyzers; and
- Output: merged test results.

TA *Results Verifier:* It verifies all returned test results. Usually test results with high confidence are treated as correct test results. Those test results that cannot be verified are sent back to TA analyzer for re-analyzing.

- Input: merged test results; and
- Output: verified test results, unverified test results.

Validated TA *Results:* They save and share all validated test results. The validated test results are categorized according the number of components.

Part V Test Database: It not only saves test results from Testing Processing, but also saves analyzed test results from AR and TA. Only validated test results can be saved in Test Database. All saved test results are shared and can be reused. Different from traditional databases, the saved test results are categorized by type and the number of components. For instance, 2-way and 3-way `F` configurations are saved in its own table respectively. Due to the large number of test results, only the roots of `X`, and `F` configurations are saved in test database. For example, configuration (a, b, c, d, e) is `F` and configuration (a, b, c) is the faulty root, so only configuration (a, b, c) is saved in `F` data table. Test results of those configurations that contain configuration (a, b, c) are automatically considered as fault.

Part VI Candidate Test Workloads Recommendation: It is used to figure out those priority configurations for testing. Based on the existing test results, it recommends those potential faults in the candidate set. Those configurations in candidate set that have one or two Hamming Distance between existing faulty configurations are recommended for TA and AR. TA, AR and Recommendation system communicate often. TA and AR send their analyzed test results to Recommendation system. Recommendation system sends related candidate configurations to them. Comparing TA and AR, the communication between Test Engine and Recommendation system is one-way direction. Only Recommendation system sends candidate configurations to Test Engine. The parent sets of faulty configurations found by AR are recommended to Test Engine for testing.

Confidence: Confidence is used to measure the reliability of each configuration's test result. Confidence (C) is the ratio of the number of one type test result (T) in all returned test results (AT) of one configuration. $C(T_i) = \frac{T_i}{AT} = \frac{T_i}{\sum_{i=1}^{n} T_i}$, and

$\sum_{i=1}^{n} C(T_i) = 1$, i is number of different types T. The confidence $C(T_i)$ measures are bounded by the interval [0,1].

For example, one test workload is processed on three virtual machines a, b, and c respectively. Machine a, b, and c return m, n, and m as results respectively. Three tests have two test results m and n. Two machines return m, about 66.6% of all test results. Test result m that has higher confidence than result n is treated as the verified test result. If test results cannot be finalized, those combinations must be tested again, until the their test result can be finalized.

6.3 TaaS as SaaS

A TaaS can be implemented as a SaaS. Similar to other SaaS systems, a TaaS database also has four layers: GUIs, workflows, services, and data. It also has three important characteristics, customization, multi-tenancy, and scalability. TaaS allows tenants to compose their TaaS applications using the existing testing services.

Figure 6.2 shows a TaaS infrastructure. It has two parts, one is the runtime platform, the other one is the customization & runtime repositories. The *Runtime platform* performs six functions.

- Scheduling: The order of TA analysis, AR analysis, and testing are scheduled according to the existing test results. New verified test results are updated and shared in a time manner.
- Provisioning: The computation resources are provided to the designated test workloads on demand.
- Monitoring: All TaaS-related activities are monitored.
- Load Balancing: Test workloads are assigned to each server according its computation capacity.

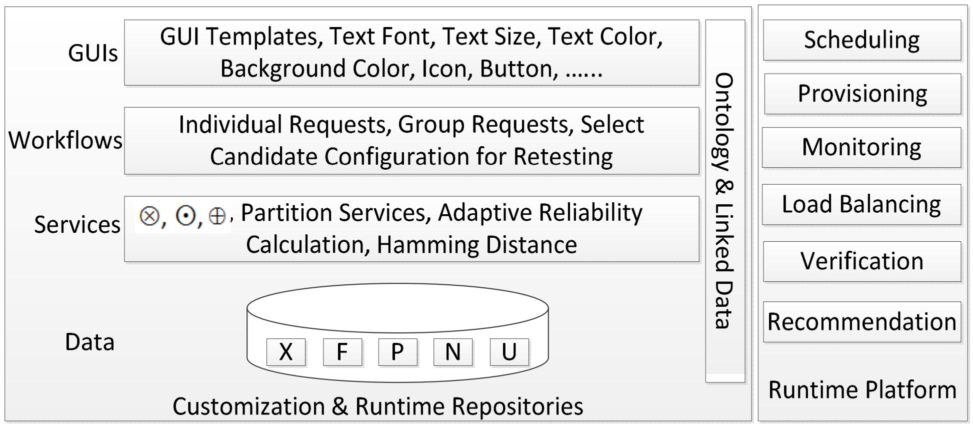

GUIs
GUI Templates, Text Font, Text Size, Text Color, Background Color, Icon, Button, ……
Workflows
Individual Requests, Group Requests, Select Candidate Configuration for Retesting
Services
⊗, ⊙, ⊕, Partition Services, Adaptive Reliability Calculation, Hamming Distance
Data
X F P N U
Customization & Runtime Repositories
Ontology & Linked Data
Scheduling
Provisioning
Monitoring
Load Balancing
Verification
Recommendation
Runtime Platform

Figure 6.2: TaaS Infrastructure

- Verification: All test results need to be validated, and only validated test results are saved.

- Recommendation: The recommendation mechanism uses algorithms to analyze candidate configurations. Then it recommends those selected candidate configurations for retesting.

The *Customization & runtime repositories* have TaaS four-layer model as traditional SaaS. Each layer provides different options for tenants to compose their own TaaS applications. The ontology & Linked Data guides the TaaS composition process.

6.3.1 GUIs

Various TaaS templates are stored in the GUI repository, tenants can build their own GUIs based on the templates. Tenants can customize the existing templates, such as modifying text font and size. Commonly use operations of changing and configuring GUI appearance, such as adding/editing/deleting icons, colors, fonts, titles in pages, menus and page-section are available.

6.3.2 Workflows

Individual request: It involves testing single configurations. TA analyzes single configurations first. If its test result can be determined from existing test results, there is no need to do any testing. Otherwise, the configuration needs to be tested.

Group request: This involves testing of multiple configurations. The following procedures shows the steps.

1. Partition the space: Due to heavy workloads, the workloads are partitioned and assigned to different Test Engines for processing. One configuration may be assigned to multiple Test Engines for redundant testing. The partitioning process intelligently adjusts the workloads according to the computation capacity of each Test Engine.

2. Evaluation operations $\otimes$ and $\odot$: The assigned workloads should be analyzed by TA first. The test results are saved in the Test Database.

3. Merge operation $\oplus$: $\oplus$ operation is used to merge testing results from different Test Engines.

4. Store consistent results: When it merges testing results from different Test Engines. Reliability of these results is computed, and only highly confident results are stored in the database.

5. Send for retesting: Those configurations with uncertain test results will be sent back for further testing.

Select Candidate Configurations for Retesting: While a large number of configurations needs to be tested, but the number of faulty configurations is only a small percentage of configurations. It is difficult to find these faulty configurations. `NU` configurations are added into testing consideration for increasing the chance of finding faulty configurations. `NU` configurations are from `N` configurations, but are treated as `U` configurations. Based on the existing `F` configurations, different algorithms such as Random, Hamming Distance, and mixed strategies can be used to get `NU` configurations from `N` configurations Wu *et al.* (2014).

6.3.3 Services

$\otimes$ *Operation Service:* It is used to get the test result of $V(\mathcal{T}_1 \cup \mathcal{T}_2)$ from $V(\mathcal{T}_1)$ and $V(\mathcal{T}_2)$.

$\odot$ *Operation Service:* Test result of one configuration can be composed by merging tests results of all its interactions, such as $V(\mathcal{T}) = \bigodot_{\mathcal{I} \subseteq \mathcal{T}} V(\mathcal{I})$, where $\mathcal{I}$ is an interaction covered by configuration $\mathcal{T}$.

$\oplus$ *Operation Service:* It merges testing results from different Test Engines.

Partition Service: It partitions test workloads and assigns them to different Test Engines, according to the computation capacity of each Test Engine.

Adaptive Reliability Calculation of Configurations and Processors Service: It calculates reliability of all returned test results. Similarly, those processors that always return correct test results are treated as reliable processors. The test results from reliable processors have higher reliability that others.

Hamming Distance Service: It gets those `NU` configurations from `N` configurations by calculating Hamming distance based on `F` configurations. Usually `NU` configurations have one or two Hamming distance from `F` configurations.

6.3.4 Runtime Composition, Execution and Scalability

Composition: Assuming the GUI layer has five components, each has three options, as $GUI\ template_1$, $GUI\ template_2$, $GUI\ template_3$. The workflow layer has three components, individual request, group request, and select candidate configura-

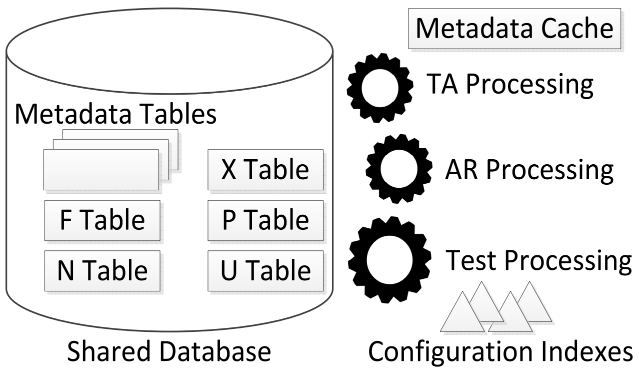


Figure 6.3: Database Integration

tion for retesting. The service layer has six components: $\otimes$, $\odot$, $\oplus$, partition service, adaptive reliability calculation, and Hamming distance. The data layer has five types of data, `X`, `F`, `P`, `N`, and `U`.

Two tenants, $Tenant_1$ and $Tenant_2$, use these components to compose their own applications. $Tenant_1$ chooses *GUI template*$_1$, *text font*$_2$, *text size*$_2$, individual request, $\otimes$, $\odot$, $\oplus$, `F`, `P`, and `U`. $Tenant_2$ chooses *GUI template*$_3$, *text color*$_2$, *background color*$_1$, group request, select candidate configuration for retesting, $\otimes$, $\odot$, $\oplus$, partition service, adaptive reliability calculation, all data types.

Tenant Application Execution: When a tenant request comes in, the TaaS will see if the tenant application is in the memory. If it is, the tenant application will be called. If the tenant application is not in the memory, the tenant application metadata will be retrieved so that tenant application components can be retrieved from the database, the tenant application will be composed, and then compiled, the executable code will be deployed to a processor.

Assuming, three processors are available, $Tenant_1$'s TaaS processes can be executed in one machine. $Tenant_2$'s TaaS processes group requests. More test workloads and five test result statuses are involved in group requests. Partition service splits the workloads into three parts. Each part is executed on different machines.

As same configurations may be tested by multiple processors, all returned test results need to be checked by adaptive reliability calculation.

Scalability: The load balancer will assign different workloads to balance the processors. Each processor has stateless servers. Workloads at processors can also be migrated to another processor to resume computation. Furthermore, the shared database allow each processor to access the data.

6.4 Experimental Results

A group of simulations have been performed, and this section provides one SaaS example for testing. The SaaS has four layers, and each layer has five components, and each component has two options as the initial settings. When the current workloads are finished reaching to 20%, one new component is added to each layer until each layer has ten components. The experiments are done for t-way configurations for $2 \leq t \leq 6$.

The initial settings of infeasible, faulty, and irrelevant configurations are shown in Table 6.2. The number of candidate configurations from five components to ten components each layer are also shown in Table 6.2. When new components are added, the infeasible, faulty, and irrelevant rate are 2%, 0.0003%, and 3% respectively. There are total eight VMs with same computation capacity in this simulation. It supposes that the maximum computation capacity of each VM is 50,000,000 configurations. There two thresholds, $threshold_{min}$ is 20,000,000 configurations (20% of the maximum), and $threshold_{max}$ is 35,000,000 configurations (75% of the maximum). When the workloads of each VM is greater than $threshold_{max}$, new VM will be assigned. When the workloads of each VM is less than $threshold_{min}$, workloads of this VM will be assigned to others, and this VM will stop working. It assumes that when 20% of current workloads are finished, new component will be added to each layer until each layer has ten components.

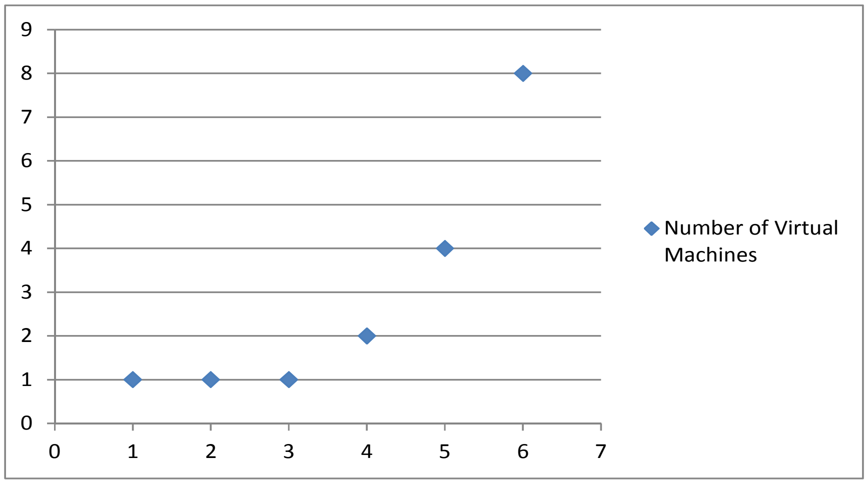


Figure 6.4: The Number of Virtual Machines of Each Attempt

There are six attempts in total from five components to ten components of each layer. Figure 6.4 shows the number of VMs used in each attempt. When the number of components in each layer increases, the trend is that more VMs are required to process the workloads. From five components to seven components, only one VM is used. The number of components in each layer increases to eight, nine, and ten, the corresponding numbers of VMs are two, four, and eight respectively. Figure 6.5 shows the average computation time of each VM in each attempt. The computation time is counted in seconds. Similarly, when the workloads increase, more execution time of each VM spends. From six to seven components, there is one big gap between the execution times of two VMs. Since only one VM is used to process six or seven components, more workloads are added when six component increases to seven components, more execution times are spent. When nine components increase to ten, the average computation time slightly decreases. Since four more VMs are used when it has ten components in each layer, the average workloads of each VM decreases and its corresponding execution time also decreases.

Based on the proposed TaaS design, when workloads increase, the current working mechanism can be extended. More VMs are added, including test engines, TA

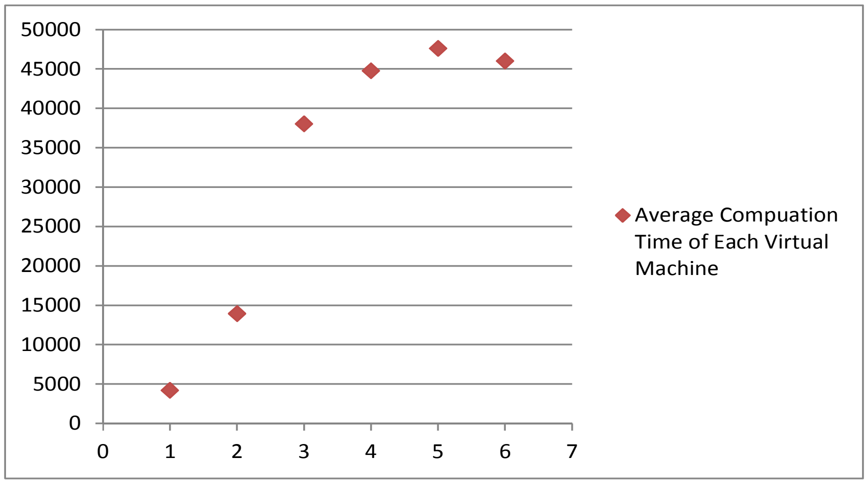


Figure 6.5: Average Computation Time of Each Virtual Machine of Each Attempt

Analyzers, AR Analyzers. TaaS scalability issues involving redundancy and recovery, and data migration can be solved in the proposed design. The returned test results from each VM can be shared to other VMs through the current test results sharing mechanism.

6.5 Conclusion

This paper talks about TaaS architecture and design. New issues introduced by cloud are discussed and three generations of TaaS are proposed. A TaaS framework has been proposed. TaaS as one type of SaaS can be used to test SaaS. This paper illustrates the process of using TaaS to test SaaS.

Table 6.1: SaaS and TaaS Comparison

	SaaS	**TaaS**
Automated Provisioning	SaaS automatically adjusts the computing resources following the change of workloads. It scales up with increasing loads with automated expansion. Vice verse, it scales down.	TaaS supports automated provisioning and de-provisioning of computing resources in a scalable cloud test environment.
Migration	Each unit of data can be moved for scalability.	The migration process can be monitored, traced, and tested.
Automated Load Balancing	It automatically balances the workloads across multiple virtual machines.	Balanced testing workloads are assigned to different servers.
Composition	The complicated services are composed by basic functional services.	Based on the candidate test workloads, the specific TaaS is also composed by basic testing services.
Concurrent	The workloads of SaaS can be executed concurrently on different servers.	Testing workloads can be distributed to different servers and processed at the same time.
Crash and Recovery	When crash happens, SaaS can be recovered from backup copies on different servers.	TaaS uses backup copies to recover from crash automatically.

Table 6.2: The Initial Settings of Configurations

	Size								
w.	`X`	`F`	`N`	5 Com	6 Com	7 Com	8 Com	9 Com	10 Com
2	5	15	20	760	1,104	1,512	1,984	2,520	3,120
3	50	5	200	9,120	16,192	26,208	39,680	57,120	79,040
4	500	0	2,000	77,520	$1.70*10^5$	$3.28*10^5$	$5.75*10^5$	$9.42*10^5$	$1.46*10^6$
5	5,000	0	$2*10^4$	$4.96*10^5$	$1.36*10^6$	$3.14*10^6$	$6.44*10^6$	$1.21*10^7$	$2.11*10^7$
6	$5*10^4$	1	$2*10^5$	$2.48*10^6$	$8.61*10^6$	$2.41*10^7$	$5.80*10^7$	$1.25*10^8$	$2.46*10^8$

Chapter 7

INTEGRATED FAULT DETECTION AND TEST ALGEBRA FOR COMBINATORIAL TESTING IN TAAS (TESTING-AS-A-SERVICE)

7.1 Framework

7.1.1 TA *and* AR *Relationship*

Figure 4.1 shows the relationship between TA and AR Tsai *et al.* (2014). AR identifies `P` and `F` configurations, as well as fault locations. TA detects `X` or `F` configurations using identified `X` or `F` configurations. AR prefers `P`, as if a configuration is `P`, all the interactions (from 2-way to t-way) within the configuration have status of `P`, and thus many interactions can be eliminated from consideration. But TA prefers `F` and `X` as they can eliminate many configurations.

Figure 5.1 shows another relationship between TA and AR. The test database that contains `X`, `F`, `P`, `N`, `U` tables is shared by both TA and AR, and they can do their own tasks concurrently. Specifically, multiple AR tasks can be run at the same time with multiple TA tasks to reduce the execution time.

An interesting scenario will happen when an `X` or `F` is identified, if there are few `U` configurations available, AR will not be effective as it needs a collection of configurations. One way to do this is to convert some `N` configurations into `NU` configuration, and run AR on `U` and `NU` configurations Wu *et al.* (2014). In this case, it is possible that a configuration is evaluated before it has been selected. Furthermore, multiple strategies are available to identify those `N` for `NU`.

Figure 7.1 shows the procedures of processing `N` and `U` configurations. Once `NU` is selected, `U'` is the set after those `NU` configurations are removed from `U`. `NU` and `U'`

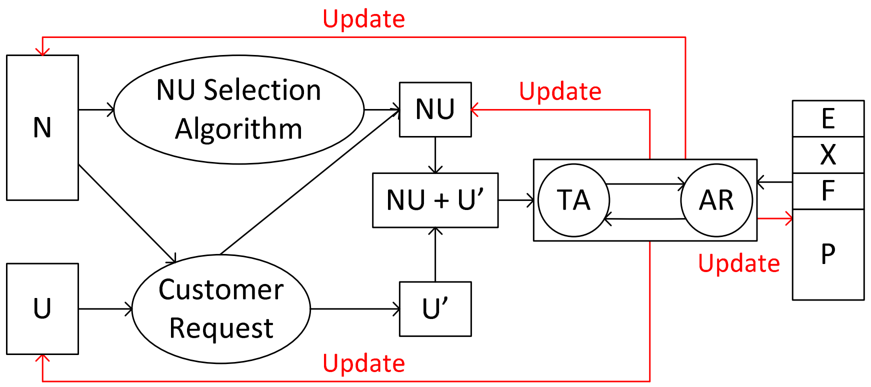


Figure 7.1: TaaS Testing Framework for Processing `N` and `U` Configurations

are used for analysis. During the process, `N` is continuously being updated whenever new components are added (`N` is increased) and/or another `NU` selection is made (`N` is decreased).

Multiple `NU` selection algorithms are available:

- Random Algorithm (NU_r)

  Random algorithm is easy to implement, but it does not use any existing test result for picking `NU` configurations. Thus, the configurations may not be optimal for testing purpose.

- User-Hint Future Configuration (NU_{fc})

  This is based on users input. In many cases, users know about their needs and know those configurations that are useful. Users can also make their decisions based on the existing test results. As this is a manual process, the quality of this process highly dependent on the experience level of the involved users.

- Linkage/Weight (NU_w)

  Linkage/Weight method finds related configurations in `N` based on existing test results. For example, configuration (a, b, d) may be selected if configuration (a, b, c) has been found to be faulty as (a, b, d) is close to (a, b, c). The usage of each configuration can be used to select NU configurations. For example, if configurations (a, b, c), (b, c, d) are frequently used, configuration (a, b, d) and (a, c, d) can be selected for NU as they are related to frequently used configurations.

The NU_w Linkage Algorithm is below used to find NU_w configurations from N configuration set.

Algorithm 6 NU_w Linkage Algorithm

```
Input:
    frequently used configurations
Output:
    $NU_w$ configurations
1: Find the overlap among different configurations
2: if overlap is found then
3:     Calculate the subsets of the overlap
4:     Calculate the possible combinations of non-overlaps
5:     Compose new configurations with subsets and non-overlaps combinations
6: end if
7: if the composed configuration is in N configuration set then
8:     Return the composed configuration
9: end if
```

- Identify X/F (NU_f) for AR and TA

  Different from NU_w, NU_f is used to identify those potential candidate X/F configurations in N. NU_f configurations are closely related to existing X/F configurations. It analyzes existing X/F configurations and finds those components that are often in X/F configurations. Then it uses the components to compose

NU_f configurations. The idea is that a configuration close to a faulty configuration may be faulty too as TA will be productive only if a F combination is identified. Thus, from time to time, AR may choose to select a configuration that is likely to fail to identify a faulty combination to enhance TA performance.

- Identify P (NU_p) for AR

  Similar as NU_f, NU_p is used to identify those potential candidate P by searching those closely related to existing P configurations. It analyzes existing P configurations and finds those components are often in those P configurations. Then it uses the components to compose NU_p configurations. The idea is that a configuration close to a P configuration is likely to a P configuration too. In AR algorithm, a P configuration is useful as it can eliminate many combinations from testing, and thus a version of AR seek to maximize the probability of identifying P configuration before testing.

- The relationship between NU and N

  In a CT project, it may have a large number of N configurations. In that case, NU size will be much smaller than N size. The goal is to select those NU configurations that can optimize the testing process. Sometimes it is for P configurations (to optimize AR), and sometimes for F configurations (to optimize TA). The relationships among NU and N are as follows: $|\mathtt{NU}| << |\mathtt{N} - \mathtt{NU}| < |\mathtt{N}|$.

  The best way to run AR is to eliminate as many X/F configurations first, and this can be done by TA.

- The best way to run AR + TA together

  AR may need X/F configurations to be identified by TA before testing, and TA needs an X/F combinations to be identified to be productive. Thus these two processes may need to wait for one another. The waiting time affects the efficiency. As the problem size is large, it is possible to run AR and TA in parallely working on different portions of the problem size. AR / TA performs its own tasks concurrently, update the shared database, and merge results. The TA rules guarantee that the merged results will be consistent.

7.1.2 Integrated Process

Based the previous discussion, the incremental and integrated process is proposed as shown in Figure 7.2. In the framework, AR runs test configurations to find those `F` configurations first. AR analysis stops until all `F` configurations are identified. The identified analyzed `F` configurations are used by TA to eliminate those `X`, `F`, and `N` configurations from candidate configuration set. TA analysis stops until all `X`, `F`, and `N` configurations are eliminated. After candidate configuration set is analyzed by AR and TA, new components are added. Then TA analyzes the candidate configuration set with new components to eliminate those `X`, `F`, and `N` configurations according to existing test results. After that, AR analyzes the candidate configuration set using existing test results. The same analysis process is repeated until all 6-way interactions are analyzed. The process stops when the number of `N` configurations equals to zero, and all `X` and `F` configurations are identified.

Different number of `U` configurations are sent to each processor to do TA analysis according to its computation capacity. When one processor finishes analyzing the assigned `U` configurations, new `U` configurations will be assigned. New `U` configurations are randomly picked from candidate configuration set. The finalized configurations will be updated in test database. Those configurations that cannot be finalized through testing analysis will be tested. The process stops until all `U` configurations are finalized.

Algorithm 7 AR & TA Integrated Processing Algorithm

```
Input:
    X, F, N, P, U configurations
Output:
    deduced F configurations, updated U configurations
 1: while N == 0 && all X, F configurations are identified && no new components do
 2:     if F configuration exist then
 3:         Run AR
 4:         Return identified F interaction
 5:     end if
 6:     while all related X, F, and N configurations are eliminated based on existing test results do
 7:         if F interaction | X configuration | N configuration exists && U configuration exists then
 8:             Run TA
 9:             Return identified F configurations & updated U configurations
10:         end if
11:         Add new components
12:     end while
13: end while
```

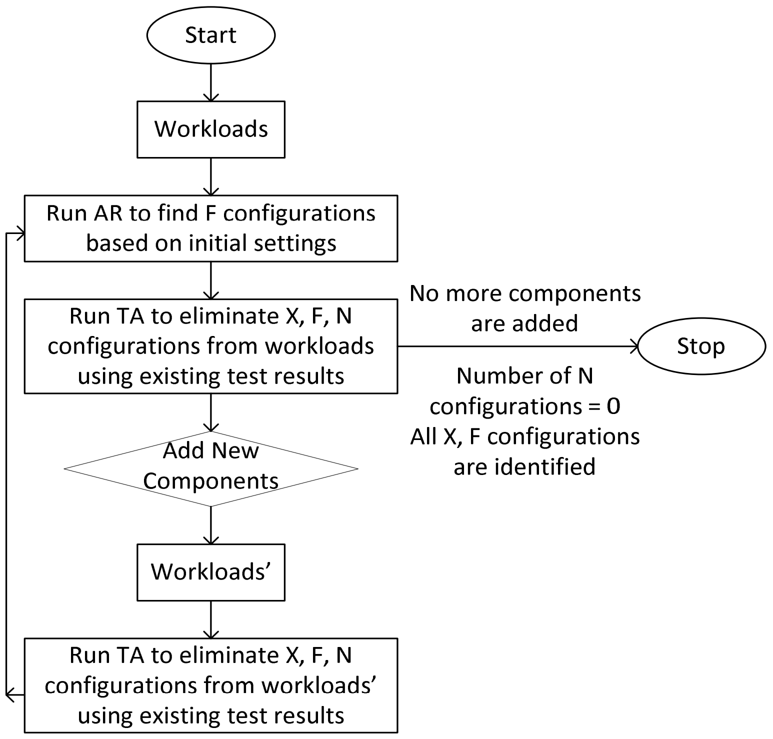

Start
Workloads
Run AR to find F configurations based on initial settings
Run TA to eliminate X, F, N configurations from workloads using existing test results
No more components are added
Stop
Number of N configurations = 0
All X, F configurations are identified
Add New Components
Workloads'
Run TA to eliminate X, F, N configurations from workloads' using existing test results

Figure 7.2: The Flowchart of AR and TA Analysis

The incremental process is proposed to emulate the SaaS tenant application development process. After a SaaS system is deployed, new tenants can be added, while other tenant applications are being executed at the same time. Each time a new tenant is added, zero or more components will be added with at least one new configuration.

It is possible to add components in a batch mode rather than on a continuous mode to save the incremental computation. It is also possible to run a non-incremental manner where all the components and all required configurations are known.

7.1.3 Framework Illustration

One example is used to illustrate the overall framework. There are six components (a, b, c, d, e, f) and each component has two options. There are total 2^6 configurations. Suppose 2-way combination (c, e) and 6-way combination (a, b, c, d, e, d) are faulty.

Table 7.1: 6-Component Example Initial Settings

	Configurations #	`F` #	`X` #	`N` #	`P` #
2-way	15	1	0	3	2
3-way	20	0	0	5	3
4-way	15	0	0	0	2
5-way	6	0	1	0	1
6-way	1	1	0	0	0

Table 7.2: Related Configurations Eliminated by TA

	`F` #	`X` #	`N` #	Configurations Need to be Tested
2-way	1	0	3	9
3-way	4	0	5	8
4-way	6	0	0	7
5-way	4	1	0	0
6-way	1	0	0	0

The initial settings are shown in Table 7.1, involving `X`, `N`, and `P` configurations.

Step 1: AR analyzes the workloads, based on the initial settings. One test hits 6-way failure and four tests hit 2-way failures. AR stops until all `F` configurations are identified.

Step 2: Based on the initial settings and the identified `F` configurations, TA analyzes the candidate set. Four `F` 3-way configurations, six `F` 4-way configurations, and four `F` 5-way configurations are eliminated by TA as shown in Table 7.2. TA stops until all related `X`, `F`, and `N` configurations are eliminated.

Step 3: One new component g is added and workloads increase. Suppose one `F` 5-way configuration, one `F` 6-way configuration, and two `N` 4-way configurations are also added in the increased configurations. The increased number of configurations

Table 7.3: Increased Configurations by Adding New Component

	Increased Configuration #
2-way	6
3-way	15
4-way	20
5-way	15
6-way	6

Table 7.4: Related Configurations Eliminated by TA

	`F` #	`N` #	Configurations Need to Be Tested
2-way	0	0	6
3-way	1	0	14
4-way	4	2	14
5-way	6	0	9
6-way	0	0	6

are shown in Table 7.3.

Step 4: Based on the existing test results, TA analyzes the candidate configuration set. One `F` 3-way configuration, four `F` 4-way configurations, and six `F` 5-way configurations are eliminated as shown in Table 7.4. TA stops until all related `X`, `F`, and `N` configurations are eliminated.

Step 5: Based on existing test results, AR analyzes those configurations in candidate configuration set. Four test cases hit 5-way failure and three test cases hit 6-way failure. AR stops until all `F` configurations are identified.

Step 6: Based on the existing test results and the new identified `F` configurations, TA analyzes the candidate configuration set. One `F` 5-way configuration and three `F` 6-way configurations are eliminated by TA as shown in Table 7.5. TA stops until all

Table 7.5: Related Configurations Eliminated by TA

	F #	Configurations Need to Be Tested
2-way	0	6
3-way	0	14
4-way	0	14
5-way	1	8
6-way	3	3

related X, F, and N configurations are eliminated.

Step 7: No more components are added. All X and F configurations are eliminated from candidate configuration set. And no N configuration is in the candidate set. The integrated process stops.

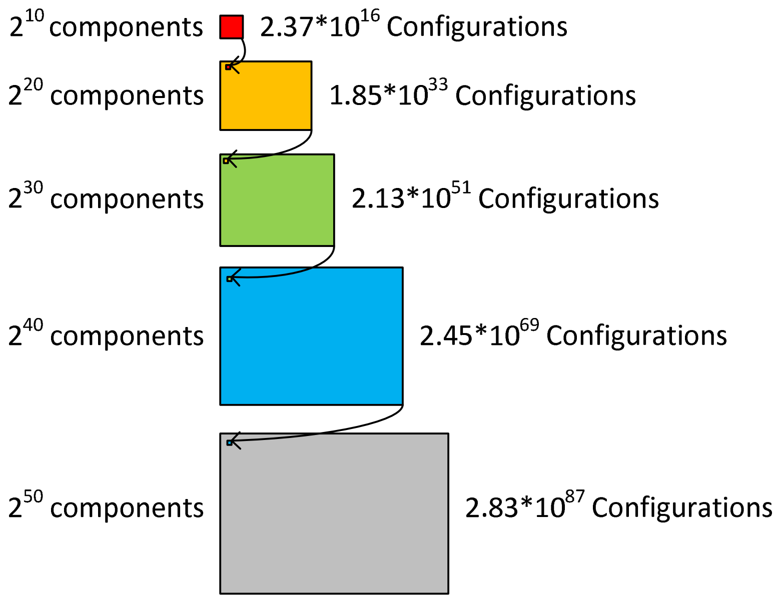
7.2 Experiments and Results

7.2.1 Experiment Setup

To evaluate the integrated process including its scalability, we performed extensive simulations, this section provides five large experiments with 2^{10}, 2^{20}, 2^{30}, 2^{40}, and 2^{50} components, and each component has two options. The corresponding number of configurations are $2^{2^{10}} = 2^{1024} = 1.79*10^{308}$, $2^{2^{20}}$, $2^{2^{30}}$, $2^{2^{40}}$, and $2^{2^{50}}$, furthermore $2^{2^{10}} \ll 2^{2^{20}} \ll 2^{2^{30}} \ll 2^{2^{40}} \ll 2^{2^{50}}$.

Tables 7.6 to 7.10 show the number of components and configurations. For example, 2^{50} components will have $2.83*10^{87}$ 6-way configurations. Compare to previously reported experiments Kuhn (2010); Kuhn *et al.* (2008); Wu *et al.* (2014), this may be the largest experimentation size in CT known to the authors as 2014.

To visualize the growth of configurations, Figure 7.3 shows the increased workloads when the number of components increases, each time the total number of configura-



2^10 components
2.37*10^16 Configurations
2^20 components
1.85*10^33 Configurations
2^30 components
2.13*10^51 Configurations
2^40 components
2.45*10^69 Configurations
2^50 components
2.83*10^87 Configurations

Figure 7.3: The Number of Components in TaaS Simulation

tions increases exponentially. Based on the initial seeded faults, each configuration from 2-way to 6-way (up to 2.83*10^{87}) is analyzed.

Table 7.6, 7.7, 7.8, 7.9, 7.10 show the number of all involved configurations from 2-way to 6-way, initial setting of `X`, `N`, and `P` configurations respectively for 2^{10}, 2^{20}, 2^{30}, 2^{40}, and 2^{50} components.

Note that `F` configurations have been set up for a challenging situation where few faults are seeded for 3-way to 6-way interactions, and in some cases no fault is seeded, especially in the light of the enormous size of t-way configurations in these systems. For example, out of 2.45*10^{59}, only one fault is seeded as a 6-way fault in 2^{40} system. In this case, AR will need to test many configurations to encounter a failure, and as few `F` configurations are available, TA will not be efficient in eliminating configurations.

In the simulation, 500 candidate configurations are sent to each processor one time. When 450 configurations are processed, another 500 candidate configurations are added to each processor. Figure 7.4 shows the nature of concurrent AR and TA tasks in the integrated process. At the beginning, only AR can execute as it

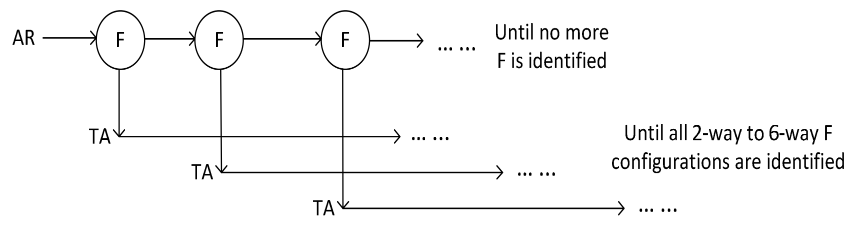

AR
F
F
F
… …
Until no more F is identified
TA
… …
TA
… …
TA
… …
Until all 2-way to 6-way F configurations are identified

Figure 7.4: The Integrated Process

needs to identify F or X first. When AR identifies a failure, the faulty interactions can be quickly identified by reasoning, and once the faulty interaction is identified, TA will initiate a new concurrent process to eliminate related faulty interactions automatically. When AR detects the second failure and second fault interactions, TA may not have completed its execution, and thus another TA process is initiated based on the newly identified faulty interactions. In this way, numerous TA processes can be executed at the same time with the AR process. Multiple AR processes can be executed too, but each takes different configuration for testing. Each TA process will stop when all related F configurations from 2-way to 6-way are identified. As all these processes share the same database, any update done by any concurrent TA processes will be available to the AR process immediately, TA rules ensures that any results obtained by TA will be eventually consistent regardless if the same configuration may be identified multiple times. For example, combination (a, b) is faulty, so is (c, d), then configuration (a, b, c, d) will be identified by two TA processes, one from (a, b), the other (c, d), but both processes will produce consistent results.

Furthermore, as the number of components increases, the number of F, X, N, and P configurations also increases, and for the same number of components, the number of X, N, and P configurations increases from 2-way to 6-way.

Another important consideration is that each initial F and X seeded are unique. For example, if combination (a, b) is a seeded 2-way fault, then (a, b, c) cannot be an initial 3-way fault for any component c, nor it can be a 3-way X combination. By

Table 7.6: Initial 2^{10} Components Experiment Setups

	Configs #	`F` Configs by AR	`X` Configs	`N` Configs	`P` Configs
2-way	523,776	10	524	348,311	149,276
3-way	$5.35*10^8$	2	$5.35*10^5$	$3.51*10^8$	$1.49*10^8$
4-way	$2.73*10^{11}$	0	$2.73*10^8$	$1.78*10^{11}$	$7.70*10^{10}$
5-way	$9.29*10^{13}$	0	$9.29*10^{10}$	$6.07*10^{13}$	$2.61*10^{13}$
6-way	$2.37*10^{16}$	1	$2.37*10^{13}$	$1.54*10^{16}$	$6.64*10^{15}$

Table 7.7: Initial 2^{20} Components Experiment Setups

	Configs #	`F` Configs by AR	`X` Configs	`N` Configs	`P` Configs
2-way	$5.50*10^{11}$	10,486	$5.50*10^8$	$3.59*10^{11}$	$1.55*10^{11}$
3-way	$1.92*10^{17}$	4	$1.92*10^{14}$	$1.25*10^{17}$	$5.51*10^{16}$
4-way	$5.03*10^{22}$	1	$5.03*10^{19}$	$3.29*10^{22}$	$1.41*10^{22}$
5-way	$1.06*10^{28}$	1	$1.06*10^{25}$	$6.93*10^{27}$	$2.97*10^{27}$
6-way	$1.85*10^{33}$	1	$1.85*10^{30}$	$1.20*10^{33}$	$5.20*10^{32}$

arranging the initial `F` and `X` seeded in this manner, these can be detected by AR only, not by TA. Otherwise, if (a, b, c) is also seeded, TA will pick it up when it detects that (a, b) is a `F`, and eliminated it automatically. Thus, the initial `X` and `F` seeded have been carefully designed so that they can be detected by AR only to evaluate the integrated process under a challenging situation.

All simulations are run on four Intel Xeon processor E7-4870 v2 (30M Cache, 2.30 GHz, 15 cores) machines. Then machines run for about a month on a dedicated mode for the experiments.

7.2.2 Experiment Results

The following results are obtained:

Table 7.8: Initial 2^{30} Components Experiment Setups

	Configs #	`F` Configs by AR	`X` Configs	`N` Configs	`P` Configs
2-way	$5.76*10^{17}$	$1.07*10^{7}$	$5.76*10^{15}$	$3.76*10^{17}$	$1.62*10^{17}$
3-way	$2.06*10^{26}$	10	$2.06*10^{24}$	$1.34*10^{26}$	$5.81*10^{25}$
4-way	$5.54*10^{34}$	2	$5.54*10^{32}$	$3.61*10^{34}$	$1.56*10^{34}$
5-way	$1.19*10^{43}$	2	$1.19*10^{41}$	$7.71*10^{42}$	$3.34*10^{42}$
6-way	$2.13*10^{51}$	1	$2.13*10^{49}$	$1.39*10^{51}$	$6.02*10^{50}$

Table 7.9: Initial 2^{40} Components Experiment Setups

	Configs #	`F` Configs by AR	`X` Configs	`N` Configs	`P` Configs
2-way	$6.04*10^{23}$	$1.10*10^{10}$	$6.04*10^{21}$	$3.94*10^{23}$	$1.70*10^{23}$
3-way	$2.22*10^{35}$	23	$2.22*10^{33}$	$1.44*10^{35}$	$6.32*10^{34}$
4-way	$6.09*10^{46}$	4	$6.09*10^{44}$	$3.98*10^{46}$	$1.71*10^{46}$
5-way	$1.34*10^{58}$	2	$1.34*10^{56}$	$8.75*10^{57}$	$3.75*10^{57}$
6-way	$2.45*10^{69}$	1	$2.45*10^{67}$	$1.60*10^{69}$	$6.93*10^{68}$

- Each experiment has been run 3 times, and all the results are presented with the average of three runs;
- All 2-way to 6-way faults seeded have been identified by AR in all these experiments;
- All the identified 2-way to 6-way faults have been used by TA to eliminate as many corresponding configurations;
- All the experiments have been conducted using incremental process as stated in Section 7.1 until there is no more `N` configurations, i.e., the system runs out of new configuration for testing.
- Hundreds of thousand TA analysis run concurrently.

Table 7.10: Initial 2^{50} Components Experiment Setups

	Configs #	`F` Configs by AR	`X` Configs	`N` Configs	`P` Configs
2-way	$6.34*10^{29}$	$1.13*10^{13}$	$6.34*10^{27}$	$4.14*10^{29}$	$1.78*10^{29}$
3-way	$2.38*10^{44}$	35	$2.38*10^{42}$	$1.55*10^{44}$	$6.67*10^{43}$
4-way	$6.70*10^{58}$	5	$6.70*10^{56}$	$4.38*10^{58}$	$1.88*10^{58}$
5-way	$1.51*10^{73}$	3	$1.51*10^{71}$	$9.87*10^{72}$	$4.23*10^{72}$
6-way	$2.83*10^{87}$	1	$2.83*10^{85}$	$1.84*10^{87}$	$8.04*10^{86}$

- $N_c = N_{xta} + N_{far} + N_{fta} + N_{nta} + N_p + N_u$. N_c is the total number of t-way configurations ($2 \leq t \leq 6$). N_{xta} is the number of `X` t-way configurations eliminated by TA. N_{far} and N_{fta} are the number of `F` t-way configurations identified by AR and eliminated by TA respectively. N_{nta} is the number of `N` t-way configurations eliminated by TA. N_p is the number of `P` t-way configurations. N_u is the number of `U` t-way configurations.

Figure 7.5 shows the number of test configurations need to identify all t-way faults ($2 \leq t \leq 5$) by AR. Note that these is different from those configurations eliminated by TA. These initial seeded faults can be identified by AR only due to the unique design described in the previous subsection. As few faults are seeded in 3-way to 6-way interactions, the numbers of test configurations needed for AR are large.

7.2.3 Measurements

N_{tc}: **the number of t-way configurations needed to identify a t-way fault, and versus the total number of t-way configurations.** Table 7.11 and Table 7.12 show the number of configurations from 2-way to 6-way in identifying faults and the related percentage over corresponding configurations from 2-way to 6-way. For example, in the case 2^{10} 2-way interaction faults, AR needs to perform

2,340 tests on average to hit ten failures, and this is only 0.45% of all 2-way possible configurations. While the number of configurations needed increases with increasing components, but the percentage becomes smaller rapidly. In fact, only $2.54*10^{-67}$% of 3-way configurations will be needed. If one compares the ratio but using the total number of configurations (rather than 3-way configurations), the ratio will be even smaller. In general, AR needs to perform a tiny fraction of the total number of t-way configurations of $2 \leq t \leq 6$.

Number of t-way interaction eliminated: TA eliminated any 2-way to 6-way F configurations caused by seeded 2-way to 6-way faults, and this is shown in Table 7.13. For each fault identified, a huge number of configurations are eliminated on average. For example, while only 2 faults are seeded in 3-way interactions in 2^{30} system, $6.19*10^{24}$ configurations are eliminated by TA. This shows that while AR needs to perform many tests to identify an interaction fault, but each interaction fault identified can lead to significant reduction in overall test workloads. As the number of components increases, the reduction is even more significant. The total number of configurations eliminated by the integrated process are shown in Table 7.17. Similarly, the X configurations eliminated by TA and their ratio to the total number t-way configurations are shown in Table 7.16.

Table 7.18 shows the number of configurations that need to be tested and the corresponding percentage of the total number of configurations, and the results showed that consistently only about 1.6% of configuration need to be tested, or about 98.4% of configurations do not needed to be tested.

An interesting question is about the consistency of 1.6% as similar results were obtained using different parameters on smaller scale systems Tsai *et al.* (2014); Wu *et al.* (2014). This percentage does not go down or up significantly regardless of the experiment size.

Figure 7.6 shows the total workloads and testing workloads saved plotted using a logarithmic graph. Based on the initial settings, the number of components from 2^{10} to 2^{50} have more 98.3% deduction rate that is shown in Figure 7.7.

R_{ec}: **the ratio of the eliminated configurations over total number of configurations.** Table 7.16 shows TA deduction efficiency. When the number of components increases, the efficiency always keeps at the same level. The deduction rate increases from 2-way configurations to 6-way configurations.

The deduction rate with `N` configurations is higher than the deduction rate without `N` configurations. `N` configuration is one key factor to affect the TA efficiency. More `N` configurations have, TA analysis is more efficient.

Computational complexity needed to eliminate those faulty t-way configurations for $2 \leq t \leq 6$: All related `F` or `X` configurations up to 6-way are eliminated from testing considerations by TA analysis, according to the initial `F` or `X` seeded interactions. Regardless of the initial `F` or `X` seeded interactions, the worst case is that all the 2-way to 6-way configurations must be visited. For n components, the number of all configurations from 2-way to 6-way is $C_2^n + C_3^n + C_4^n + C_5^n + C_6^n$ with time complexity O(n^6). All `F` and `X` configurations from 2-way to 6-way are identified by TA within time complexity O(n^6). While these numbers look large, but when one compares them to the total number of possible configuration, these numbers are actually small as O(n^6) $\ll$ O(2^n). When n increases, the percentage of n^6 over 2^n decreases. $\lim_{n \to infty} \frac{n^6}{2^n} = 0$.

Table 7.15 shows all the computation steps that of TA performed in the simulation to eliminate 2-way to 6-way configurations for 2^{10} to 2^{50} systems, as well as the ratio of the computation steps over the total configuration. One can see that the ratio is almost zero for all these systems, with the largest number being $4.35*10^{-275}$ and it is close to zero already. As TA deals with configuration elimination, not components,

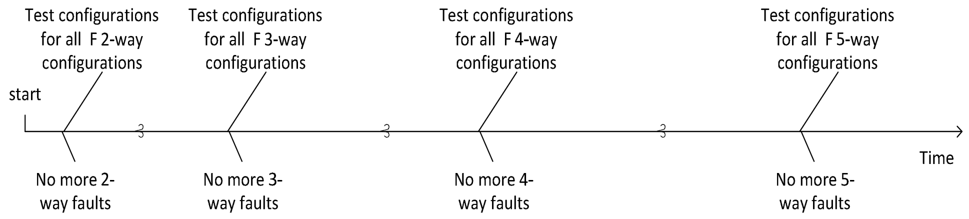

Test configurations for all F 2-way configurations
Test configurations for all F 3-way configurations
Test configurations for all F 4-way configurations
Test configurations for all F 5-way configurations
start
Time
No more 2-way faults
No more 3-way faults
No more 4-way faults
No more 5-way faults

Figure 7.5: AR Test Configurations to Identify All T-way Faults ($2 \leq T \leq 5$)

Table 7.11: Test Configurations # of Identifying Faults in 2-way to 6-way Configurations

Components	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}
	Test Configurations				
2-way	2,340	$4.39*10^6$	$6.67*10^9$	$9.10*10^{12}$	$1.17*10^{16}$
3-way	1,334	5,816	24,370	82,156	$1.66*10^5$
4-way	na	4,285	6,941	78,896	40,755
5-way	na	12,138	19,871	57,666	38,376

thus TA needs to traverse a tiny fraction of the total number of configuration to cover most 2-way to 6-way configurations.

7.3 Conclusion

With the arrival of cloud computing, the need to perform large CT to identify faulty interactions and configurations, instead of just coverage, has also arrived. At the same time, the cloud also provided significant computing resources including C-PUs and storage that allow people to perform CT exercises that were not possible before. This paper has proposed a TaaS framework that allows large CT exercises to detect faulty interactions and configuration in SaaS. The proposed framework combines faulty detection with asynchronous TA to eliminate related configurations concurrently. The goal of this project is to demonstrate that it is possible to run large CT with a huge number (2^{50}) of components with $2^{2^{50}}$ of configurations. This may be

Table 7.12: Percentage of Test Configurations Over 2-way to 6-way Configurations

Components	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}
	Percentage (%)				
2-way	0.45	0.000798	$1.16*10^{-6}$	$1.51*10^{-9}$	$1.85*10^{-12}$
3-way	0.000249	$3.03*10^{-12}$	$1.18*10^{-20}$	$3.70*10^{-29}$	$6.97*10^{-38}$
4-way	na	$8.52*10^{-18}$	$1.25*10^{-29}$	$1.30*10^{-40}$	$6.08*10^{-53}$
5-way	na	$1.15*10^{-22}$	$1.67*10^{-37}$	$4.30*10^{-52}$	$2.54*10^{-67}$

Table 7.13: `F` Configurations Deduction by TA

Components	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}
	Deducted `F`				
2-way	10	10,486	$1.07*10^{7}$	$1.10*10^{10}$	$1.13*10^{13}$
3-way	10,222	$1.10*10^{10}$	$1.15*10^{16}$	$1.21*10^{22}$	$1.27*10^{28}$
4-way	$5.22*10^{6}$	$5.76*10^{15}$	$6.19*10^{24}$	$6.65*10^{33}$	$7.14*10^{42}$
5-way	$1.77*10^{9}$	$2.01*10^{21}$	$2.22*10^{33}$	$2.44*10^{45}$	$2.68*10^{57}$
6-way	$4.52*10^{11}$	$5.27*10^{26}$	$5.95*10^{41}$	$6.70*10^{56}$	$7.54*10^{71}$

the largest CT experiments known to the authors as 2014 with $2.45*10^{69}$ 6-way configurations alone. The combined process has been simulated using 60 CPUs that run for almost a month on a dedicated mode with a large number of concurrent processes. The process successfully eliminated about 98.4% of test configurations from testing consideration consistent across these experiments. These exercises demonstrated that the proposed TaaS framework can work on large project with large number of components and configurations.

Table 7.14: `X` Configurations Deduction by TA

Components	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}
	Deducted `X`				
2-way	524	$5.50*10^{8}$	$5.76*10^{15}$	$6.04*10^{21}$	$6.34*10^{27}$
3-way	$2.08*10^{7}$	$6.79*10^{15}$	$6.59*10^{24}$	$6.92*10^{33}$	$7.85*10^{42}$
4-way	$1.27*10^{10}$	$2.32*10^{21}$	$2.12*10^{33}$	$2.27*10^{45}$	$2.49*10^{57}$
5-way	$4.44*10^{12}$	$5.10*10^{26}$	$5.31*10^{41}$	$5.41*10^{56}$	$5.97*10^{71}$
6-way	$1.25*10^{15}$	$9.79*10^{31}$	$8.19*10^{49}$	$9.26*10^{67}$	$1.12*10^{86}$

Table 7.15: Computation Steps of TA Analysis

Compos	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}
Configs	$2^{2^{10}}$	$2^{2^{20}}$	$2^{2^{30}}$	$2^{2^{40}}$	$2^{2^{50}}$
Steps	$7.79*10^{30}$	$5.54*10^{64}$	$7.68*10^{100}$	$8.68*10^{136}$	$1.28*10^{173}$
Ratio	$4.35*10^{-278}$	approx 0	approx 0	approx 0	approx 0

Table 7.16: TA Deduction Rate

Components	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}
2-way	96.32%	96.32%	96.33%	96.33%	96.33%
3-way	97.44%	97.44%	97.45%	97.45%	97.45%
4-way	98.15%	98.15%	98.15%	98.16%	98.16%
5-way	98.31%	98.31%	98.32%	98.32%	98.33%
6-way	98.36%	98.37%	98.37%	98.37%	98.37%

Table 7.17: Configurations Eliminated by AR & TA

Number of Components	Configurations Eliminated	Percentage
2^{10}	$2.33*10^{16}$	98.34%
2^{20}	$1.82*10^{33}$	98.34%
2^{30}	$2.10*10^{51}$	98.38%
2^{40}	$2.41*10^{69}$	98.39%
2^{50}	$2.78*10^{87}$	98.39%

Table 7.18: Configurations Need to Be Tested

Number of Components	Configurations Need to Be Tested	Percentage
2^{10}	$3.93*10^{14}$	1.66%
2^{20}	$3.14*10^{31}$	1.66%
2^{30}	$3.45*10^{49}$	1.62%
2^{40}	$3.94*10^{67}$	1.61%
2^{50}	$4.56*10^{85}$	1.61%

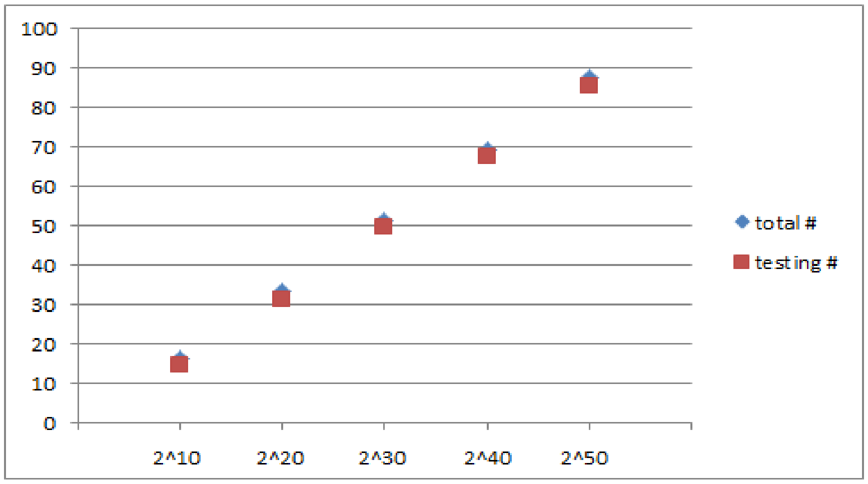


Figure 7.6: Total Workloads and Testing Workloads Saved

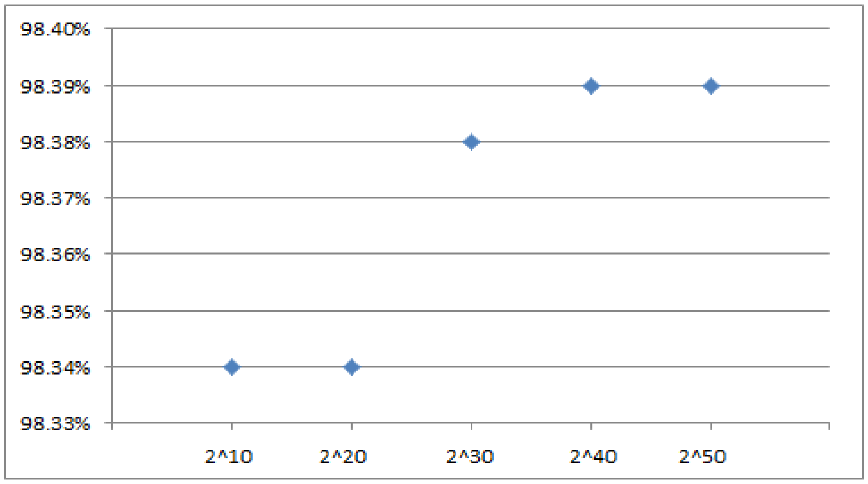


Figure 7.7: Configuration Deduction Rate

Chapter 8

CONCLUSION

With the arrival of cloud computing, the need to perform large CT to identify faulty interactions and configurations, instead of just coverage, has also arrived. At the same time, the cloud also provided significant computing resources including CPUs and storage that allow people to perform CT exercises that were not possible before.

Existing CT methods mainly focus on test coverage. But high test coverage does not equal to cover more possible combinations. Actually a large number of combinations are not tested by existing CT methods. My thesis proposes an efficient way TA to explore the untested combinations in combinatorial testing. TA is one testing analysis method that analyzes existing test results to eliminate those infeasible, faulty, and irrelevant configurations from testing consideration for increasing testing efficiency. The proposed TA that defines five statuses X, F, P, N, U with a priority and three operations $\otimes$, $\odot$, and $\oplus$, is formalized by mathematic model and provides a foundation for concurrent combinatorial testing. The commutativity and associativity of defined TA operations and the efficiency of TA are proved by mathematic method. By using the TA operations, many combinatorial tests can be eliminated as the TA identifies those interactions that need not be tested. Also TA defined operation rules allow merging 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. Different simulations designed for TA rules also prove TA is an efficient way to increase testing efficiency.

The faulty root of F configuration is helpful for TA analysis. AR analyzes the faulty roots and the faulty roots are used by TA in analyzing candidate configurations. AR and TA cooperate to analyze candidate configurations for increasing testing efficiency. Based on AR and TA, a TaaS framework that allows large CT exercises to detect faulty interactions and configuration in SaaS is proposed. The proposed framework combines faulty detection with asynchronous TA to eliminate related configurations concurrently. The combined process has been simulated on a dedicated mode with a large number of concurrent processes in cloud environment. The process successfully eliminated about 98.4% of test configurations from testing consideration consistent across these experiments. These exercises demonstrated that the proposed TaaS framework can work on large project with large number of components and configurations.

REFERENCES

Ahmed, B. S. and K. Z. Zamli, “A Review of Covering Arrays and Their Application to Software Testing”, Journal of Computer Science **7**, 9, 1375–1385 (2011).

Arcuri, A. and L. Briand, “Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing”, IEEE Trans. Softw. Eng. **38**, 5, 1088–1099 (2012).

Arcuri, A., M. Z. Iqbal and L. Briand, “Formal Analysis of the Effectiveness and Predictability of Random Testing”, in “Proceedings of the 19th international symposium on Software testing and analysis”, ISSTA ’10, pp. 219–230 (ACM, New York, NY, USA, 2010).

Bai, X., 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)”, pp. 1–12 (Irvine, CA, USA, 2011).

Borazjany, M., L. Yu, Y. Lei, R. Kacker and R. Kuhn, “Combinatorial Testing of Acts: A Case Study”, in “Proceedings of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)”, pp. 591–600 (2012).

Brownlie, R., J. Prowse and M. S. Padke, “Robust Testing of AT&T PMX/StarMAIL using OATS”, AT&T Technical Journal **7**, 3, 41–47 (1992).

Bryce, R. C. and C. J. Colbourn, “One-test-at-a-time heuristic search for interaction test suites”, in “Genetic and Evolutionary Computation Conference (GECCO), Search-based Software Engineering track (SBSE)”, pp. 1082–1089 (2007).

Bryce, R. C. and C. J. Colbourn, “Expected time to detection of interaction faults”, Journal of Combinatorial Mathematics and Combinatorial Computing (to appear).

Bryce, R. C., C. J. Colbourn and D. R. Kuhn, “Finding Interaction Faults Adaptively Using Distance-Based Strategies”, in “Proceedings of the 2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems”, ECBS ’11, pp. 4–13 (IEEE Computer Society, Washington, DC, USA, 2011).

Bryce, R. C., Y. Lei, D. R. Kuhn and R. Kacker, “Combinatorial Testing”, Handbook of Software Engineering Research and Productivity Technologies (2010).

Burr, K. and W. Young, “Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage”, in “Proceedings of the Intl. Conf. on Software Testing Analysis and Review”, pp. 503–513 (West, 1998).

Calvagna, A. and A. Gargantini, “IPO-s: Incremental Generation of Combinatorial Interaction Test Data Based on Symmetries of Covering Arrays”, in “Proceedings of the International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW ’09”, pp. 10–18 (2009).

Cohen, D., 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).

Cohen, D. M., S. R. Dalal, A. Kajla and G. C. Patton, “The Automatic Efficient Test Generator (AETG) System”, in “Proceedings of International Conference on Testing Computer Software”, (1994).

Cohen, D. M., S. R. Dalal, J. Parelius and G. C. Patton, “The Combinatorial Design Approach to Automatic Test Generation”, IEEE Software **13**, 5, 83–88 (1996a).

Cohen, D. M., S. R. Dalal, J. Parelius and G. C. Patton, “The Combinatorial Design Approach to Automatic Test Generation”, IEEE Software **13**, 5, 83–88 (1996b).

Colbourn, C. J., “Covering Arrays and Hash Families”, in “Information Security, Coding Theory and Related Combinatorics”, edited by D. Crnkovic and V. D. Tonchev, vol. 29 of *NATO Science for Peace and Security Series - D: Information and Communication Security* (IOS Press, 2011).

Colbourn, C. J., S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood and J. L. Yucas, “Products of Mixed Covering Arrays of Strength Two”, Journal of Combinatorial Designs **14**, 14, 124C138 (2006).

Dalal, S. R., A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton and B. M. Horowitz, “Model-based Testing in Practice”, in “Proceedings of the 21st international conference on Software engineering”, ICSE ’99, pp. 285–294 (ACM, New York, NY, USA, 1999).

Dalal, S. R. and C. L. Mallows, “Factor-covering Designs for Testing Software”, Technometrics **40**, 234–243 (1998).

Duran, J. W. and S. C. Ntafos, “An Evaluation of Random Testing”, IEEE Trans. Softw. Eng. **10**, 4, 438–444 (1984).

Gao, J., X. Bai and W. T. Tsai, “Cloud Testing-Issues, Challenges, Needs and Practice”, in “Software Engineering: An International Journal (SEIJ), IGI Global”, vol. 1, pp. 9–23 (2011a).

Gao, J., 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 (2011b).

Ghazi, S. and M. Ahmed, “Pair-wise Test Coverage Using Genetic Algorithms”, in “Proceedings of 2003 Congress on Evolutionary Computation (CEC03)”, vol. 2, pp. 1420–1424 (2003).

Grindal, M., J. Offutt and S. F. Andler, “Combination Testing Strategies: A Survey”, Software Testing, Verification, and Reliability **15**, 167–199 (2005a).

Grindal, M., J. Offutt and S. F. Andler, “Combination Testing Strategies: a Survey.”, Softw. Test., Verif. Reliab. **15**, 3, 167–199 (2005b).

Hedayat, A. S., N. J. A. Sloane and J. Stufken, *Orthogonal Arrays: Theory and Applications* (Springer-Verlag, New York, 1999).

Huang, R., X. Xie, T. Y. Chen and Y. Lu, "Adaptive Random Test Case Generation for Combinatorial Testing.", in "COMPSAC", pp. 52–61 (IEEE Computer Society, 2012).

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

Kuhn, R., "Combinatorial Testing", `http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf/` (2010).

Kuhn, R., Y. Lei and R. Kacker, "Practical combinatorial testing: Beyond pairwise", IT Professional **10**, 3, 19–23 (2008).

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

Lei, J., "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing", `http://csrc.nist.gov/groups/SNS/acts/documents/ipo-nist.pdf/` (2005).

Lei, Y. and K.-C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing", in "Proceedings of 3rd IEEE International Symposium on High-Assurance Systems Engineering", HASE '98, pp. 254–261 (1998).

Mandl, R., "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing", Communications of ACM **28**, 10, 1054–1058 (1985).

Mathur, A. P., *Foundations of Software Testing, 2nd Edition* (Pearson Education, Upper Saddle River, New Jersey, 2013).

Muller, T. and D. Friedenberg, "Certified Tester Foundation Level Syllabus", Journal of International Software Testing Qualifications Board (2007).

Nie, C. and H. Leung, "A Survey of Combinatorial Testing", ACM Comput. Surv. **43**, 2, 11:1–11:29 (2011).

Porter, A. A., 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).

Rajappa, V., A. Birabar and S. panda, "Efficient Software Test Case Generation Using Genetic Algorithm Based Graph Theory", `www.cs.uofs.edu/~bi/2008f-html/se516/peddachappali.ppt/` (2008).

Riungu, L. M., O. Taipale and K. Smolander, "Software testing as an online service: Observations from practice", in "Proceedings of ICST Workshops", pp. 418–423 (2010).

Schroeder, P. J., P. Bolaki and V. Gopu, "Comparing the Fault Detection Effectiveness of N-way and Random Test Suites", in "Proceedings of the 2004 International Symposium on Empirical Software Engineering", ISESE '04, pp. 49–59 (IEEE Computer Society, Washington, DC, USA, 2004).

Shakya, K., T. Xie, N. Li, Y. Lei, R. Kacker and R. Kuhn, "Isolating Failure-Inducing Combinations in Combinatorial Testing Using Test Augmentation and Classification", in "Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation", ICST '12, pp. 620–623 (IEEE Computer Society, Washington, DC, USA, 2012).

Shiba, T., T. Tsuchiya and T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing", in "Proceedings of the 28th Annual International Conference on Computer Software and Applications (COMPSAC2004)", vol. 1, pp. 72–77 (2004).

Sinnema, M. and S. Deelstra, "Classifying Variability Modeling Techniques", Inf. Softw. Technol. **49**, 7, 717–739 (2007).

Tai, K.-C. and Y. Lei, "A Test Generation Strategy for Pairwise Testing", IEEE Trans. Software Eng. **28**, 1, 109–111 (2002).

Tsai, W.-T., C. Colbourn, J. Luo, G. Qi, Q. Li and X. Bai, "Test Algebra for Combinatorial Testing", in "Proceedings of the 2013 8th International Workshop on Automation of Software Test (AST)", pp. 19–25 (2013a).

Tsai, W.-T., 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 (2012).

Tsai, W.-T., Y. Huang and Q. Shao, "Testing the Scalability of SaaS Applications", in "Proceedings of IEEE International Conference on Service-Oriented Computing and Applications (SOCA)", pp. 1–4 (Irvine, CA, USA, 2011).

Tsai, W.-T., 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 (2010a).

Tsai, W.-T., 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)", (2013b).

Tsai, W.-T., J. Luo, G. Qi and W. Wu, "Concurrent Test Algebra Execution with Combinatorial Testing", in "Proceedings of 8th IEEE International Symposium on Service-Oriented System Engineering (SOSE2014)", (2014).

Tsai, W.-T., 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, 2010b).

van Lint, J. H. and R. M. Wilson, *A Course in Combinatorics* (Cambridge University Press, 1992).

Wikipedia, "All-pairs Testing", `http://en.wikipedia.org/wiki/All-pairs_testing/` (2014a).

Wikipedia, "Backtracking", `http://en.wikipedia.org/wiki/Backtracking/` (2014b).

Wikipedia, "Black-box Testing", `http://en.wikipedia.org/wiki/Black-box_testing/` (2014c).

Wikipedia, "Genetic Algorithm", `http://en.wikipedia.org/wiki/Genetic_algorithm/` (2014d).

Wikipedia, "Latin Square", `http://en.wikipedia.org/wiki/Latin_square/` (2014e).

Wikipedia, "Orthogonal Array", `http://en.wikipedia.org/wiki/Orthogonal_array/` (2014f).

Wikipedia, "White-box Testing", `http://en.wikipedia.org/wiki/White-box_testing/` (2014g).

Williams, A. and R. Probert, "A Practical Strategy for Testing Pair-wise Coverage of Network Interfaces", in "Proceedings Seventh International Symposium on Software Reliability Engineering", pp. 246–254 (1996).

Wu, W., W.-T. Tsai, C. Jin, G. Qi and J. Luo, "Test-Algebra Execution in a Cloud Environment", in "Proceedings of 8th IEEE International Symposium on Service-Oriented System Engineering (SOSE2014)", (2014).

Yilmaz, C., 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, pp. 45–54 (ACM, New York, NY, USA, 2004).

Yu, L., Y. Lei, R. Kacker and D. Kuhn, "ACTS: A Combinatorial Test Generation Tool", in "Proceedings of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)", pp. 370–375 (2013).

Zhang, J., Z. Zhang and F. Ma, *Automatic Generation of Combinatorial Test Data* (Springer, New York, 2014).

APPENDIX A

THE PROOFS OF TA DEFINED OPERATIONS

A.1 COMMUTATIVITY OF $\otimes$

The commutativity of binary operation $\otimes$.

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

Proof. Since the binary operation $\otimes$ is defined as

$\otimes$	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

Because the above matrix is symmetric on the main diagonal, the value of $V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)$ is always the same as $V(\mathcal{T}_2) \otimes V(\mathcal{T}_1)$. Thus, the commutativity of $\otimes$ holds. □

A.2 ASSOCIATIVITY OF $\otimes$

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 $V(\mathcal{T}_1) = \texttt{X}$, then according to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \texttt{X} \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \texttt{X}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = (\texttt{X} \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \texttt{X} \otimes V(\mathcal{T}_3) = \texttt{X}$. 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 $V(\mathcal{T}_1) = \texttt{F}$, 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 $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \texttt{F} \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \texttt{F}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = (\texttt{F} \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \texttt{F} \otimes V(\mathcal{T}_3) = \texttt{F}$. 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 $V(\mathcal{T}_1) = \texttt{N}$, 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 $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \texttt{N} \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = \texttt{N}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = (\texttt{N} \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \texttt{N} \otimes V(\mathcal{T}_3) = \texttt{N}$. 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 $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = V(\mathcal{T}_1) \otimes \texttt{U} = \texttt{U}$, $(V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3) = \texttt{U} \otimes V(\mathcal{T}_3) = \texttt{U}$. Thus, in this case, $V(\mathcal{T}_1) \otimes (V(\mathcal{T}_2) \otimes V(\mathcal{T}_3)) = (V(\mathcal{T}_1) \otimes V(\mathcal{T}_2)) \otimes V(\mathcal{T}_3)$. □

A.3 COMMUTATIVITY OF $\oplus$

The commutativity of binary operation $\oplus$.

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

Proof. Since the binary operation $\oplus$ is defined as

$\oplus$	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

Because the above matrix is symmetric on the main diagonal, the value of $V_1(\mathcal{T}) \oplus V_2(\mathcal{T})$ is always the same as $V_2(\mathcal{T}) \oplus V_1(\mathcal{T})$. Thus, the commutativity of $\otimes$ holds. $\square$

A.4 ASSOCIATIVITY OF $\oplus$

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 $\mathtt{E}$. 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 $\mathtt{E}$, 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 $\mathtt{X}$ and the other is not, or (b) one of them is $\mathtt{P}$ and the other is $\mathtt{F}$.

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

(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 $\mathtt{E}$ or $\mathtt{F}$. 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 $\mathtt{E}$ or $\mathtt{P}$. 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 $\mathtt{E}$, 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 $\mathtt{X}$. 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 $V_1(\mathcal{T}) = \texttt{F}$, 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 $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \texttt{F} \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \texttt{F}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \texttt{F} \oplus V_3(\mathcal{T}) = \texttt{F}$.

(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 $V_1(\mathcal{T}) = \texttt{P}$, 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 $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \texttt{P} \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \texttt{P}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \texttt{P} \oplus V_3(\mathcal{T}) = \texttt{P}$.

(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 $V_1(\mathcal{T}) = \texttt{U}$, 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 $V_1(\mathcal{T}) \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \texttt{U} \oplus (V_2(\mathcal{T}) \oplus V_3(\mathcal{T})) = \texttt{U}$ and $(V_1(\mathcal{T}) \oplus V_2(\mathcal{T})) \oplus V_3(\mathcal{T}) = \texttt{U} \oplus V_3(\mathcal{T}) = \texttt{U}$.

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

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

A.5 DISTRIBUTIVITY OF ⊗ OVER ⊕

The distributivity of binary operation $\otimes$ over $\oplus$ supporting status E.

⊗	E	X	F	P	N	U
E	E	E	E	E	E	E
X	E	X	X	X	X	X
F	E	X	F	F	F	F
P	E	X	F	U	N	U
N	E	X	F	N	N	N
U	E	X	F	U	N	U

$V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) \succeq (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2))$.

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

(1) $V(\mathcal{T}_1)$ is E. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{E}$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{E}$, and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{E}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2))$.

(2) $V(\mathcal{T}_1)$ is not E and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is E. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{E}$.

a) If one of $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ is E, then according to the operation table of $\otimes$ and $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{E}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{E}$.

b) If $V(\mathcal{T}_1) = \texttt{X}$, and both $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ are not E, then according to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{X}$, and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{X}$. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{X}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) \succ (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2))$.

c) If $V(\mathcal{T}_1)$ is not X, one of $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ is X, then the other one is F, P, N, or U. Without loss of generality, suppose that $V_1(\mathcal{T}_2) = \texttt{X}$, according to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{X}$, and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)$ can be F, N, or U. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{E}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{E}$.

d) If $V(\mathcal{T}_1) = \texttt{F}$, and both $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ are not E and X, According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{F}$, and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{F}$. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{F}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) \succ (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2))$.

e) If $V(\mathcal{T}_1)$ is not X and F, one of $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ is F, then the other one is P. Without loss of generality, suppose that $V_1(\mathcal{T}_2) = \texttt{F}$ and $V_2(\mathcal{T}_2) = \texttt{P}$, according to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{F}$, and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)$ can be N, or U. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{F}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) \succ (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2))$.

(3) $V(\mathcal{T}_1)$ is X and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is not E. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{X}$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{X}$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{X}$. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{X}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{X}$.

(4) $V(\mathcal{T}_1)$ is not E and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2) = \texttt{X}$. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{X}$. Since $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2) = \texttt{X}$, both $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ are X. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{X}$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{X}$. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{X}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{X}$.

(5) $V(\mathcal{T}_1)$ is F and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is not E and X. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{F}$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{F}$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{F}$. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{F}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{F}$.

(6) $V(\mathcal{T}_1)$ is not E and X and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is F. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{F}$. Since $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2) = \texttt{F}$, at least one of $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ is F. Without loss of generality, suppose that $V_1(\mathcal{T}_2) = \texttt{F}$, then according to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{F}$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)$ can only be F, N and U. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{F}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{F}$.

(7) $V(\mathcal{T}_1)$ is N and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is not E, X and F. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{N}$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2) = \texttt{N}$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2) = \texttt{N}$. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{N}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{N}$.

(8) $V(\mathcal{T}_1)$ is not E, X and F and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is N. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{N}$. Since $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2) = \texttt{N}$, both $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ are N. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{N}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{N}$.

(9) $V(\mathcal{T}_1)$ is P or U, and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is not E, X, F and N. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \texttt{U}$. Since $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is not E, X, F and N, $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ can only be P, N or U, and at most one of them is N. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)$ can only be N or U, and at most one of them is N. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{U}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \texttt{U}$.

(10) $V(\mathcal{T}_1)$ is not E, X, F and N, and $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is P or U. According to

the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = \mathtt{U}$. Since $V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)$ is P or U, $V_1(\mathcal{T}_2)$ and $V_2(\mathcal{T}_2)$ can only be P, N or U, and at most one of them is N. According to the operation table of $\otimes$, $V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)$ and $V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)$ can only be N or U, and at most one of them is N. According to the operation table of $\oplus$, $(V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \mathtt{U}$. Thus, $V(\mathcal{T}_1) \otimes (V_1(\mathcal{T}_2) \oplus V_2(\mathcal{T}_2)) = (V(\mathcal{T}_1) \otimes V_1(\mathcal{T}_2)) \oplus (V(\mathcal{T}_1) \otimes V_2(\mathcal{T}_2)) = \mathtt{U}$. $\square$

BIOGRAPHICAL SKETCH

Guanqiu Qi was born in Nanjing, Jiangsu Province, People's Republic of China. He attended Nanjing University, where he earned the bachelor of science in computer science in 2008.

Guanqiu further pursued his doctoral degree in the area of Software-as-a-Service (SaaS) and Testing-as-a-Service (TaaS) at Arizona State University (ASU) under the supervision of Professor Wei-Tek Tsai. He published 12 papers in top conferences and journals.