# COMBINATORIAL INTERACTION TESTING-BASED DAILY BUILD PROCESS

by
GÜLSÜM UZER

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of the requirements for the degree of
Master of Science

Sabancı University
September 2020

COMBINATORIAL INTERACTION

TESTING-BASED DAILY BUILD PROCESS

APPROVED BY:

Assoc. Prof. Dr. Cemal Yılmaz ............................................................ [illegible]
(Thesis Supervisor)

Assoc. Prof. Dr. Hüsnü Yenigün ............................................................ [illegible]

Assoc. Prof. Dr. Hasan Sözer ............................................................ [illegible]

DATE OF APPROVAL: 02.09.2020

## ABSTRACT

COMBINATORIAL INTERACTION TESTING-BASED DAILY BUILD PROCESS

GÜLSÜM UZER

Computer Science and Engineering, MS THESIS, SEPTEMBER 2020

Thesis Supervisor: Assoc. Prof. Dr. Cemal Yilmaz

Keywords: Combinatorial Interaction Testing, Daily Build Processes, Covering Arrays, Software Quality Assurance, Software Engineering

A daily build process is a process where the latest version of a software under development is obtained from its code repository on a daily basis (typically during off-work hours), configured, built, and tested against a test suite. The ultimate goal of this process is to reveal defects in the most fundamental functionalities of the system as soon as they are introduced into the codebase, so that the turnaround time for fixing them is reduced as much as possible. In this work, we first introduce combinatorial interaction testing-based daily build process where a combinatorial object is computed to systematically test the interactions between system parameters on a daily basis. We then introduce a number of different testing strategies and empirically demonstrate that the proposed approach profoundly improves the effectiveness of the standard daily build processes.

# ÖZET

KOMBİNATORYAL ETKİLEŞİM TEST TABANLI GÜNLÜK DERLEME SÜRECİ

GÜLSÜM UZER

Bilgisayar Bilimi ve Mühendisliği, YÜKSEK LİSANS TEZİ, EYLÜL 2020

Tez Danışmanı: Assoc. Prof. Dr. Cemal Yılmaz

Anahtar Kelimeler: Kombinatoryal etkileşim testi, Kongigürasyon seçenekleri, Günlük Yapılandırma, Örtme Dizileri, Yazılım Kalite Güvencesi

Günlük derleme süreci, geliştirilmekte olan bir yazılımın en son sürümünün kod havuzundan günlük olarak (genellikle çalışma saatleri dışında) alındığı, yapılandırıldığı, derlendiği ve bir test paketine göre test edildiği bir süreçtir. Bu sürecin nihai amacı, sistemin en temel işlevlerindeki kusurları kod tabanına dahil edilir edilmez ortaya çıkarmaktır, böylece bunları düzeltmek için geri dönüş süresi mümkün olduğunca kısalır. Bu çalışmada, ilk olarak, günlük bazda sistem parametreleri arasındaki etkileşimleri sistematik olarak test etmek için bir kombinatoryal nesnenin hesaplandığı kombinatoryal etkileşim testi tabanlı günlük derleme sürecini sunuyoruz. Ardından bir dizi farklı test stratejisi ortaya koyuyoruz ve deneysel olarak önerilen yaklaşımın standart günlük derleme süreçlerinin etkinliğini derinlemesine artırdığını gösteriyoruz.

# ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my advisor Assoc. Prof. Dr. Cemal Yılmaz for his support, guidance and in-depth knowledge.

I also would like to thank Hanefi Mercan for sharing the knowledge, insightful comments and experience with me.

I am indebted to my big family who always stand by me, made me feel their endless love, support and encourage me in all areas of my life, regardless of blood ties.

*For my life*
*that comes with all kinds of difficulties and beauty*

## TABLE OF CONTENTS

# LIST OF TABLES

List of Figures

## LIST OF ABBREVIATIONS

# 1. INTRODUCTION

A daily build process is a process where the latest version of a software under development is obtained from its code repository on a daily basis (typically during off-work hours), configured, built, and tested against a test suite. The ultimate goal of this process is to reveal the defects in the most fundamental functionalities of the system as soon as they are introduced into the codebase, so that the turnaround time for fixing them is reduced as much as possible. Consequently, daily build processes have been extensively used for testing software systems (Karlsson, Andersson & Leion, 2000), (Memon, Banerjee, Hashmi & Nagarajan, 2003), (McConnell, 1996).

One thing we observe, however, is that when it comes to the systematic testing of configurable software systems, the daily build processes are often carried out by testing the same set of pre-defined configurations every day. In this thesis, we conjecture that the effectiveness of the daily build processes can significantly be improved, if the interactions between configuration options are systematically tested throughout the process.

We, in particular, introduce *combinatorial interaction testing-based daily build process*, in short *CIT-daily*. CIT-daily systematically sample the configuration space of the system and test only the selected configurations on a daily basis.

At a very high level, the CIT-daily process can be summarized as follows: 1) get the latest version of the SUD; 2) systematically sample the configuration space, which will simply produce a set of configurations to be tested; and 3) configure, build, and run the SUD's test suite on each and every configuration selected. Note that this process (i.e., the steps discussed above) is carried out every day. Note further that although we, in the thesis, are mainly concerned with configurable systems, the proposed approach can readily be applicable to other types of systems as long as these systems have multiple interacting parameters, which is basically the case in almost all the software systems. In this context, a parameter is defined as a factor that can either be directly or indirectly set and that can change the behavior of the SUD. For example, configuration options, input parameters, and events in eventbased systems can all be considered to be parameters, the interactions of which can be tested by CIT-daily (Cohen, Colbourn & Ling, 2008).

CIT-daily leverages one of the most frequently used combinatorial objects for testing, namely *covering arrays*, to systematically sample the configuration spaces (Yilmaz, Fouche, Cohen, Porter, Demiroz & Koc, 2013).

A covering array takes as input a *configuration space model*, which specifies a valid configuration space for testing. More specifically, the model includes a set of configuration options (or parameters in general), their settings, and inter-option constraints (if any, as not all possible combinations of option settings may be valid in practice). Given a configuration space model and a coverage strength $t$, a $t$-way covering array is a set of configurations, in which each possible combination of option settings for every combination of $t$ options appears at least once (Yilmaz, 2012) (Fouché, Cohen & Porter, 2009).

The justification for using $t$-way covering arrays is that (under certain assumptions) they can efficiently and effectively reveal all system failures caused by the interactions of $t$ or fewer parameters (Yilmaz et al., 2013). Furthermore, many of the empirical studies strongly suggest that a majority of the configuration option-related failures are caused by the interactions of a small number of options. That is, $t$ is quite small in practice, compared to the number of configuration options. And, when $t$ is fixed, as the number of configuration options increase, the size of a covering array represents an increasingly smaller portion of the entire configuration space. Consequently, covering arrays have been extensively used for revealing option-related failures in an efficient and effective manner (Yilmaz et al., 2013) (Cohen, Dalal, Fredman & Patton, 1997) (Czerwonka, 2008) (Yilmaz, 2012) (Cohen et al., 2008).

In this thesis, to the best of our knowledge, we for the the first time leverage covering arrays in daily build processes. To this end, we have, indeed, developed a number of strategies (Section 3.1). The first strategy simply uses the same $t$-way covering array every day, where $t$ is an input to the strategy (Section 3.1.1). The second strategy computes a different $t$-way covering array to be tested every day (Section 3.1.2). The remaining strategies aims to obtain $t'$-way coverage over time by testing a $t$-way covering array every day. In particular, the third strategy computes a number of $t$-way covering arrays every day and among these covering arrays picks the one that covers the most number of previously uncovered $t'$-tuples for testing, where both $t$ and $t'$ are inputs to the strategy (Section 3.1.3). The fourth strategy guarantees to achieve $t'$-way coverage in $n$ days by testing $t$-way covering arrays every day, where $t$, $t'$, and $n$ are inputs to the strategy (Section 3.1.4). Furthermore, as a de facto strategy (the strategy which has been frequently employed in the field), we use the default configurations of our subject applications every day for testing.

To evaluate the proposed approach we carried out a series of experiments (Section 4). In these experiments, we used well-known configurable systems as subject applications and evaluated the results in a multifaceted manner. In particular, we compared different strategies by counting distinct errors, distinct test case and error pairs, tests runs, tests failures, and test skips and by also measuring the structural code coverage obtained.

The results of these studies strongly suggest that the proposed approach significantly improves the effectiveness of the daily build process. For example, for one subject application, the fourth strategy revealed 11 distinct errors, while the de facto strategy and the first strategy revealed 0 and 4 distinct errors, respectively. For another subject application, the de facto strategy, and the first strategy, and the fourth strategy revealed 1, 2, and 7 distinct errors, respectively.

## 1.1 Contributions

The contributions of this thesis can be summarized as follows:

- introduction of the combinatorial interaction testing-based daily build process, namely CIT-daily,
- a number of strategies to further improve the effectiveness of the process,
- an extensible framework for CIT-daily, which be enhanced with alternative strategies,
- empirical evaluations on well-known configurable systems.

## 1.2 Organization Of The Thesis

The remainder of the document is organized as follows: Chapter 2 provides background information on combinatorial interaction testing and on standard daily build processes; Chapter 3 introduces the proposed approach as well as the different strategies developed; Chapter 4 presents the empirical studies; Chapter 5 discusses threats to validity; Chapter 6 discusses related work; Chapter 7 presents concluding remarks; and Chapter 8 concludes with some future work ideas.

# 2. BACKGROUND INFORMATION

This chapter gives detailed information about concepts as combinatorial interaction testing, configuration space model, traditional covering arrays, seeding, method of constructing covering array, daily build process and also coverage metrics that were used in the thesis.

## 2.1 Combinatorial Interaction Testing

Many software applications are supported from testing process by containing large number of test cases to find failures in development process but each defined test cases in suite is not effective as expected to reveal failures. Instead of test cases, parameters for them become more important to get different failures and most software applications have remarkable number of parameters to be used in predetermined their own test suites (Simos, Zivanovic & Leithner, 2019)(Ahmed, Abdulsamad & Potrus, 2015).

This case has triggered to have large combination space consisting of interaction among parameters and it needs exhaustive testing to consider each of them. However, to use each parameter in different combination with others for testing process is not practical in case of limited resources. Besides this state, each parameter also has not revealed any software failure. However, most empirical studies show that the interaction between parameters has caused different software failures (Kuhn, Kacker, Lei & Hunter, 2009) (Yilmaz, 2012).

In this perspective, combinatorial interaction testing (CIT) have emerged as a new method. It offers to get more benefit from testing process of software system by detecting different software failures as early as possible and use limited time and resources efficiently (Schmidt & Kruse, 2020). The key insight of the method is to provide combinations of parameters as input to predetermined test suite. Thus, it becomes possible to detect failures triggered by interaction of parameters at early stage with using few resources by processing smaller test suite of the subject application (Nie & Leung, 2011).

## 2.2 Configuration Space Model

Configuration space model is model file for combinatorial interaction testing. It is used in generating covering array. In general, it includes configuration options, their settings and in case of existing, it also includes valid constraints. It serves creating configurations to be used in predetermined test cases as input. If any constraints exist, the set of configurations in produced covering array do not violate this constraint. Thus, the model is created carefully. Incorrect configuration space model causes misuse of resources for testing subject application and gives inaccurate results (Yilmaz et al., 2013).

This model is typically formed by developers, domain or test experts by analyzing not only business side of the application but also source code manually. This process is important to know impact of each configuration option on subject application (Ahmed, Gargantini, Zamli, Yilmaz, Bures & Szeles, 2019). After this analysis, the model should have only crucial configuration options, not all because as the number of configuration options increase in the model by including unnecessary configuration, the efficiency of the combinatorial interaction testing method decreases. The reason of it is that the size of covering array is depend on the number of configuration defined in the model. In case of existing more configuration options in the model, more configuration will be produced for covering array. Also, there will be unnecessary or ineffective configurations.

## 2.3 Traditional Covering Array

Traditional covering array that is a set of configurations is created from the information placed in configuration space model. Each row of the covering array represents one configuration as a set of parameter values for testing process of subject application. It also called as t-way covering array and t generally refers *coverage strength* (Yilmaz, 2012) (Schmidt & Kruse, 2020). The term means that every possible combination of t options in configuration space model appears at least once in t-way covering array.

For instance, there are 2 options with setting level of 2 and 1 options with setting level of 3. In general way, there should be totally 12 configurations to cover all relationship of options. However in 2-way covering array, only 6 configurations are enough to cover all 2-way combinations of three options for exhaustive testing. It means that any selected two columns contain all combinations for specific two options placed in selected columns (Nie & Leung, 2011). This state is figured in Table 2.1 and Table2.2 .

Table 2.1 Total combinations for 3 options

| option1 | option2 | option3 |
|---|---|---|
| True | 1 | False |
| True | 1 | True |
| True | 2 | False |
| True | 2 | True |
| True | 3 | False |
| True | 3 | True |
| False | 1 | False |
| False | 1 | True |
| False | 2 | False |
| False | 2 | True |
| False | 3 | False |
| False | 3 | True |

Table 2.2 For 3 options, 2-way CA

| option1 | option2 | option3 |
|---|---|---|
| True | 1 | False |
| False | 3 | True |
| False | 2 | False |
| True | 2 | True |
| False | 1 | True |
| True | 3 | False |

In addition to them, if any constraint is defined in configuration space model, covering array does not contain configuration that violates the constraint. For instance, if there is a constraint like option1 = True => option2 = 3 , the 2-way covering array should be figured like in Table 2.3. It has already 6 configurations but any row does not represent a configuration that violates specified constraint.

Table 2.3 For 3 options, 2-way CA with constraint

| option1 | option2 | option3 |
|---|---|---|
| True | 2 | False |
| False | 3 | True |
| False | 1 | False |
| True | 1 | True |
| False | 2 | True |
| False | 3 | False |

One reason behind selecting 2-way covering array example is that 2-way or 3-way covering array is commonly used. Up to t = 6, different t-way covering arrays can be preferred to reveal failures. Most studies in the article point that around %80 of failures can be detected by 2 and 3-way covering array. While increasing t value up to 6, the percentage of success of detecting failure can be observed (Kuhn, Lei & Kacker, 2008).

## 2.4 Seeds

All configurations in covering array have not same importance for testing process. Some of them are more critical than other in order to find failures. However, any covering array cannot make this distinction and all configurations in the array are to be input for predetermined test cases by considering that they have equal priorities (Bryce & Colbourn, 2006).

To construct prioritization mechanism among configurations, seeding is used. In literature, seed points out a set of specified configurations. As mentioned earlier, covering array contains a set of configurations. In this method, a part of covering array includes specific configurations and the remaining content of it is extended from other configurations. As a result of this process, t-way covering array is incrementally created. Thus, it aims to not only guarantee that determined configurations are covered by covering array but also not consider which configuration is already tested before. With the help of it, limited resources and time are used efficiently (Yilmaz et al., 2013).

The mechanism of seed is also supported by different tools like ACTS (Yu, Lei, Kacker & Kuhn, 2013), PICT (Czerwonka, 2008), SST (Nie, Xu, Shi & Wang, 2006).

## 2.5 Method of Constructing Covering Array

There are three types of covering array. One of them is denoted by *CA(N;t,p,v)*. It is fixed size covering array that means all configurations has *v* level of settings. *p* also points out number of existing configuration options. The array size is *N* and *t* represents which way covering array is used for it. Another type is called as mixed-covering array (MCA) that is denoted by *MCA(N;t,p,(v1,v2,v3,...vp))*. Its difference from previous type of covering array is to not have same level of settings for the each configuration options. *v1,v2,v3...vp* represent that first configuration has v1 level of setting, second one has v2 level of settings and continues like in this order up to *p*th configuration options. The latest type is named as variable strength covering array (VSCA) that is denoted by *VSCA (N; t,p, (v1, v2, ..., vp) , C)*. Its size is *N* x *p* and it has t-way covering array that contains specification of covering array *C* (Memon, Porter, Yilmaz, Nagarajan, Schmidt & Natarajan, 2004) (Ahmed & Zamli, 2011).

To generate these kinds of covering array is counted as NP-Hard problem in the literature. To solve it, four different methods have been proposed and based on these methods various tools have been implemented (Kobayashi, Tsuchiya & Kikuno, 2002) (Kuliamin & Petukhov, 2011). These methods can be listed in greedy, heuristic search-based, mathematical and random search-based methods. Besides it, tools can be listed as ACTS, Jenny, PICT, SST.

In this subsection, we have not focused on mathematical detail of the methods. We already focused on Jenny tool that is developed on greedy methods because of using in the thesis.

Jenny is combinatorial interaction testing tool. By giving configuration space model in its own format, it produces t-way covering array. It has not any user interface so it can be used in command line as following line.

```
./jenny -n3 -s5 4 3 2 5 -w1abc2d -w1d2abc | sed -f jenny_gen.sed > jenny_gen.txt
```

The parameter *n* represents which t-way is used for generating covering array. Next parameter *s* is used for specifying random generator number. It enables us to have different covering array for different parameter value. If the parameter does not define in the command line, it produces covering array in default by assuming the value of s is 0. The next list *4 3 2 5* in the line represents what the setting level of each option is in order. Thus, we have observed that first option has 4 , second option has 3, third option has 2 and last option has 5 settings. *w* also points out constraint structure. *-w1abc2d* explains that while first option is value of a or b or c, second option cannot be value of d. The last part of parameter *sed* is used for defining input format. In default, Jenny use alphabetical value for the option but it is possible to define option and value name specially in file whose extension is sed. It also write produced covering array to txt file (Zimmerer, 2004).

## 2.6 Daily Build

Daily build is part of agile driven development. In this type of development, developers codes what they want to achieve in their local branch of the source code repository and sent their development to master branch of it in order to merge their development with existing code. However, it is not easy to do that. In case of huge projects, there are hundreds of request to be integrated into one and it is not possible to analyze manually which request can be appropriate for the master branch. To solve this problem by doing automatically for every day, daily build process is considered as an alternative solution (Karlsson et al., 2000).

While continuing development process depend on end user requirements, it enables developers to control the result of their developments with respect to its functionality and predetermined test suite in daily basis. The flow of it can be summarized as that the latest version of repository in current day is getting from source repository. It is packaged, compiled and built as a first stage. If this stage gives successful result, testing stage as a next step starts running. In this stage, there are some predefined test suite to run in that. They have already used in it. Then, all results related to build and test are used by developers or teams .If the first stage does not result in successful, only build result is produced and daily build is completed a failing without running testing process. Thus, any developer can easily observe which test cases are passed, what kind of errors the repository has or whether there is failure or not as a result of this process (Memon et al., 2003) (Memon & Qing Xie, 2004) (Memon, Nagarajan & Xie, 2005). The key insight of it is also to help revealing failures at early stage. If it is considered that software development is quite large and complex activity, any changes on it can cause chaotic situation and it is quite important to detect it as early as possible (Karlsson et al., 2000). Besides the insight, it also help minimizing integration risk by checking all iteration on the version located in master branch daily. Additionally, it keeps quality of source code repository to a certain level (McConnell, 1996).

## 2.7 Coverage Metrics

Coverage is a fundamental concept to evaluate efficiency of test suite and track both quality and maintenance issue for any software. It has various metrics like line, branch, decision, package, requirement and etc. but we have focused some of them that are related to the thesis. They will be explained in following subsections (Shahid, Ibrahim & Mahrin, 2011) (Grinwald, Harel, Orgad, Ur & Ziv, 1998).

### 2.7.1 Line and Instruction Coverage

Two metrics of them are instruction coverage and line coverage. We have evaluated together because they complements each other. Instruction coverage gives us information about how many code is covered or missed. While calculating the information, all code is considered as Java byte code. On the other hand, for line coverage, it is enough to be executed at least one instruction placed in the line. For this reason, instruction coverage does not considered as line coverage. They are different concept from each other. The differences can be summarized with an basic example like *System.out.println(string temp)* command. In general perspective, it can be seen as the line has one line and one instruction however Jacoco plugin considers that this command line is a single line even if the line has 3 bytecode instructions. These instructions of the line that are extracted by using *javap : java class file dissembler* command are *getstatic* using java/lang/System.out:java/io/PrintStream library, *ldc* compiling String temp and *invokevirtual* using java/io/PrintStream.println:(java/lang/String library).

### 2.7.2 Branch Coverage

Other metric is branch coverage. It counts how many branches are executed or missed and generates total number for the repository as a result. Also, Jacoco, coverage plugin used in thesis, categorizes the result in three classes. If all branches in line have been missed, it shows in red color. If some branches have been executed and remaining has not in the line, it shows in yellow color. Then, if all branches have been executed, it shows in green color in generated report.

### 2.7.3 Method Coverage

Other metric is method coverage. The logic of instruction coverage metric takes part in this metric. If at least one instruction in the method, the method is counted as being executed. Besides this, Jacoco plugin accepts that all constructors and static initializers are methods.

### 2.7.4 Complexity

The last metric is complexity pointing out cyclomatic complexity. According to its definition by Watson, Wallace & McCabe (1996) "cyclomatic complexity is the minimum number of paths that can, in (linear) combination, generate all possible paths through a method". Jacoco plugin counts all paths for each module and based on the result, calculates how many paths have been executed or missed then generates this complexity information. So it is important indication how many cases are missing for covering associated module (Watson et al., 1996) (Shepperd, 1988). All these metrics is crucial and effective way for evaluating the test results belonging to specific repository that runs in associated configuration (Mohamed, Sulaiman & Endut, 2013).

# 3. APPROACH

At a very high level, CIT-daily takes as input a software under development (SUD), a configuration space model for the SUD, which specifies the valid configuration space for testing, and depending on the strategy employed (Section 3.1), one or more coverage criteria to be satisfied (e.g., coverage strengths). It then operates as follows (Algorithm 1): 1) the latest version of the SUD is obtained from its code repository (line 2); 2) the configuration space is systematically sampled by computing a covering array with the goal of obtaining full coverage under the given criteria (line 4); and 3) for each and every configuration included in the covering array, the SUD is configured, built, and tested by running its test suite in the configuration (lines 5-8).

Note that these steps are carried out every day and that they are readily applicable to test any software systems (not just the configurable ones) as long as these systems have some interacting parameters, which can change the behaviour of the system, such as input parameters and user events. Note further that the ultimate goal of using covering arrays is to test the interactions between configuration options on a daily basis, so that option-related failures can be discovered as early as possible.

**Algorithm 1** The CIT-daily process.

```
1: procedure CIT-DAILY($SUD$, $model$, $strategy$)
2:     for each day do
3:         Download the latest version of $SUD$
4:         $ca \leftarrow strategy.run(model)$
5:         for each $cfg \in ca$ do
6:             $SUD.configure(cfg)$
7:             $SUD.build()$
8:             $SUD.test()$
```

## 3.1 Combinatorial Interaction Testing Strategies

We have developed a number of different combinatorial interaction testing (CIT) strategies to go with CIT-daily. Below, these strategies are discussed from the simplest one to the most complex one. Each enhancement was made to further improve the effectiveness of the CIT-based daily build process.

### 3.1.1 Strategy 1

Our first strategy, namely Strategy 1, takes as input a coverage strength $t$, computes a $t$-way covering array once, and use the same $t$-way covering array every day to test the SUD. Note that this is a basic strategy inspired from standard CIT practice; given $t$, test the SUD with a $t$-way covering array to (under certain assumption) reveal all failures caused by the settings of $t$ or fewer options. Our approach is different in that we do this on a daily basis by using the latest version of the SUD.

**Algorithm 2** Strategy 1

```
1: procedure Strategy1($model$, $args$)
2:     $t \leftarrow args\{t\}$
3:     $ca \leftarrow$ deterministically compute a $t$-way covering array for $model$
4:     return $ca$
```

Algorithm 2 presents this strategy. A strategy in the CIT-daily framework takes as input a configuration space model (*model* in line 1) and some strategy arguments (*args* in line 1). Note that for this work all the strategies are geared towards obtaining combinatorial coverage. Therefore, the strategy parameters are given in terms of the coverage strengths to be obtained. Note further that to have an extensible CIT-daily framework, which can support any arbitrary strategy, the framework accepts the strategy arguments in the form of key-value pairs (line 2).

As indicated in Algorithm 2, Strategy 1 takes as input a coverage strength (line 2) and computes a $t$-way covering array to be used with the daily build process (line 3).

### 3.1.2 Strategy 2

Strategy 2, as was the case with Strategy 1, takes as input a coverage strength $t$. Unlike Strategy 1, however, Strategy 2 generates a different $t$-way covering array every day for testing. Algorithm 3 presents this strategy.

The ultimate goal of this approach is to vary the configurations tested as much as possible throughout the days to reveal more defects, while guaranteeing the same basic coverage (in this case, $t$-way coverage). Note that although a different $t$-way covering array is used every day, all these covering arrays are guaranteed to cover all valid $t$-way combinations of settings. However, the higher-strength combinations (thus, the configurations) that they cover may change. With varying these "accidentally covered" combinations, these strategies aims to cover exercise different behaviors of the SUD.

**Algorithm 3** Strategy 2

```
1: procedure STRATEGY2($model$, $args$)
2:     $t \leftarrow args\{t\}$
3:     $ca \leftarrow$ randomly compute a $t$-way covering array for $model$
4:     return $ca$
```

### 3.1.3 Strategy 3

Strategy 3 takes as input two coverage strengths, namely $t_1$ and $t_2$ where $t_1 < t_2$, together with a positive integer $n$. The goal is to obtain $t_2$-way coverage over time by testing a $t_1$-way coverage every day. To this end, every day, Strategy 3 computes $n$ $t_1$-way covering arrays and then among these arrays picks the one that covers the most number of previously uncovered $t_2$-tuples in an attempt to reduce the number of days required to achieve $t_2$-way coverage. When the $t_2$-way coverage is obtained the process is repeated, every time starting from scratch. Algorithm 4 presents Strategy 3.

**Algorithm 4** Strategy 3

```
1: procedure STRATEGY3($model$, $args$)
2:     $t_1 \leftarrow args\{t_1\}$
3:     $t_2 \leftarrow args\{t_2\}$
4:     $n \leftarrow args\{n\}$
5:     $cas \leftarrow \{\}$
6:     for n times do
7:         $ca \leftarrow$ randomly compute a $t_1$-way covering array for $model$
8:         $cas \leftarrow cas \cup ca$
9:     $ca \leftarrow$ Pick the $ca$ in $cas$ with the best $t_2$-way coverage
10:    return $ca$
```

### 3.1.4 Strategy 4

Strategy 4 is similar Strategy 3, in the sense that it takes as input $t_1$ and $t_2$ ($t_1 < t_2$) and aims to achieve $t_2$-way coverage by testing a $t_1$-way covering array everyday. It differs from Strategy 3 in that the number of days in which $t_2$-way coverage needs to be obtained is taken as input, namely $k$, and Strategy 4 guarantees the coverage in exactly $k$ days. Algorithm 5 presents this strategy.

To this end, Strategy 4 first computes a $t_2$-way covering array (line 5), then divides this array into $k$ equal or almost equal-sized, non-overlapping partitions (line 6), and finally uses each part as a seed (Section 2.4) to compute a $t_1$-way covering array (lines 8-10).

Each, $t_1$-way covering array is used on a different day. Therefore, after $k$ days, it is guaranteed to obtain $t_2$-way coverage as the collection of all the configurations tested throughout the $k$ days is guaranteed to contain all the seeds used (i.e., all the configurations included in the $t_2$-way covering array initially computed).

Note that while Strategy 3 is opportunistic, one can determine the number of days it should take to obtain higher-order coverage in Strategy 4.

**Algorithm 5** Strategy 4

```
1: procedure STRATEGY4($model$, $args$)
2:     $t_1 \leftarrow args\{t_1\}$
3:     $t_2 \leftarrow args\{t_2\}$
4:     $k \leftarrow args\{k\}$
5:     $ca \leftarrow$ randomly compute a $t_2$-way covering array for $model$
6:     $seeds \leftarrow$ Divide $ca$ into (almost) equal-sized, non-overlapping partitions
7:     $cas \leftarrow \{\}$
8:     for each $seed \in seeds$ do
9:         $ca \leftarrow$ randomly compute a $t_1$-way covering array around $seed$
10:        $cas \leftarrow cas \cup ca$
11:    return $cas$
```

### 3.1.5 Implementation

We have implemented an extensible CIT-daily framework. Figure 3.1 presents the high level architecture of the framework, which is implemented in Python.

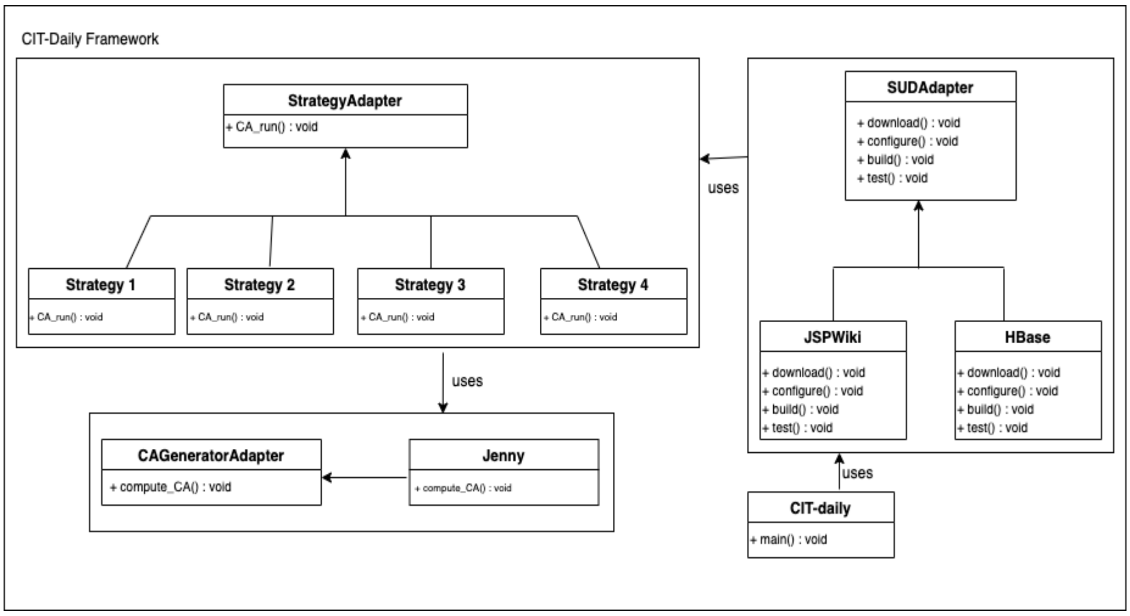

Figure 3.1 The high level architecture of the CIT-daily framework.

To ensure that the framework can work with any SUDs, covering array constructors, and CIT strategy, we have leveraged the Adapter design pattern (Ramirez & Cheng, 2010). That is, the interactions between the CIT-daily framework and the aferomentioned components are captured in the form of an adaptor interface, so that the framework can be extended with new SUDs, new covering array generators, and new strategies by simply implementing the respective interfaces.

```
interface SUD {
    void download(void);
    boolean configure(Configuration cfg);
    boolean build();
    boolean test();
    Report get_report();
}
```

Figure 3.2 The SUD adaptor interface.

```
interface CoveringArrayGeneratorAdaptor {
    CA compute_CA(ConfigurationSpaceModel model,
                  CoverageStrength t,
                  Seed seed,
                  boolean randomized)
}
```

Figure 3.3 The covering array generator adaptor interface.

```
interface StrategyAdaptor {
    CA run(ConfigurationSpaceModel model,
           StrategyArguments args)
}
```

Figure 3.4 The strategy adaptor interface.

Figures 3.2-3.4 presents these interfaces by using a Java-like syntax for readability. Furthermore, we here report somewhat simplified version of these interfaces as in the actual implementation typically more functions are included in these interfaces to handle all the low-level details.

Figure 3.2 presents the adaptor interface to be implemented for each SUD. The interface has the basic operations to download, configure, build, and test the SUD and harvest the results. In this interface, a configuration (i.e., *Configuration*) is represented as a collection of key-value pairs (i.e., configuration option-setting pairs).

Figure 3.3 presents the adaptor interface to be implemented for each covering array generator to be supported by the CIT-daily framework. The interface has an operation to compute a covering array (either in a deterministic or randomized manner) for a given configuration space model, coverage strength, and seed.

We have also implemented a number of optimization techniques. For example, our configuration space model distinguishes between *compile-time* and *runtime* options. A compile-time option is an option, the setting of which is set as a part of the build process. A runtime option, on the other hand, is an option, which is set at runtime while the SUD is running.

The CIT-daily framework recognizes static configurations and (if asked) cache these static configurations on a secondary storage, so that if the same static configuration is needed in the future, the executables can directly be obtained from the cache, rather than re-configuring and re-building the system from scratch.

Figure 3.4 presents the adaptor interface to be implemented for each CIT strategy to be supported by the CIT-daily framework. The interface has an operation to compute the covering array to be used on the current day. The interface assumes that the respective function is called once a day. Furthermore, different strategies may need different number of arguments. Therefore, in the CIT-daily framework strategy arguments are represented in the form of parameter-value pairs.

# 4. EXPERIMENTS

To evaluate the proposed approach, we have carried out a number of empirical studies.

## 4.1 Subject Applications

In these experiments, we used two frequently-used, configurable software systems, namely *Apache JSPWiki* (v2.11.0.M7) (Foundation, 2020c) and *Apache Hbase* (v2.3.0) (Foundation, 2020a).

### 4.1.1 Subjects

Apache JSPWiki is a leading WikiWiki engine. Besides supporting all traditional features of Wiki engine, JSPWiki has numerous additional features (Foundation, 2020c). It was initially released 7 years ago and since then have been constantly evolving (Foundation, 2020c). In the experiments, we used JSPWiki with Java JDK v11.

Apache HBase is a Hadoop database – a distributed, scalable, and a big data store. HBase has been evolving for 13 years (Foundation, 2020a). In the experiments, we used HBase with Java JDK v1.8.

We opted to use these applications as our subject applications because they possess characteristics that are common to configurable software systems, such as having a large user base, developed by a team of stakeholders, and evolving continuously.

### 4.1.2 Test Suites

For the subject applications, we used the test cases that were developed by the developers of these applications and that come with the source code distributions as the test suites.

More specifically, for JSPWiki, we used a toal of 1023 test cases. And, for HBase, we used between 1046 and 2110 test case, which changed during the period of experiments due to the addition of the new test cases to the code base.

### 4.1.3 Configuration Space Model

We, furthermore, read the user manuals, inspect the source code, and run small-scale experiments (as needed) to determine the configuration space model to be used for each subject application.

In particular, given the test suites of these subject applications, we attempted to choose the configuration options and settings that can affect the behavior of the test cases. And, to keep the cost of the experiments under control, we did this while reducing the number of options needed.

Table 4.1 Configuration space model used for Apache JSPWiki.

| Configuration Options | Settings | Type |
|---|---|---|
| jspwiki.diffProvider | ExternalDiffProvider, ContextualDiffProvider, TraditionalDiffProvider | compile-time |
| jspwiki.encoding | UTF-8, ISO-8859-1 | compile-time |
| jspwiki.translatorReader.matchEnglishPlurals | TRUE,FALSE | compile-time |
| jspwiki.urlConstructor | ShortViewURLConstructor, ShortURLConstructor, DefaultURLConstructor | compile-time |
| jspwiki.allowCreationOfEmptyPages | TRUE,FALSE | compile-time |
| jspwiki.breakTitleWithSpaces | TRUE,FALSE | compile-time |
| jspwiki.attachment.allowed | .png, .jpg, .zip, .jar | compile-time |
| jspwiki.login.throttling | TRUE,FALSE | compile-time |
| jspwiki.pageNameComparator.class | HumanComparator, LocalComparator | compile-time |
| jspwiki.attachment.forbidden | .html, .htm, .php, .asp, .exe | compile-time |
| jspwiki.attachment.forceDownload | .html, .htm | compile-time |
| jspwiki.searchProvider | LuceneSearchProvider, BasicSearchProvider" | compile-time |
| jspwiki.defaultprefs.template.editor | plain, WikiWizard, FCK | compile-time |
| jspwiki.defaultprefs.template.sectionediting | TRUE, FALSE | compile-time |
| jspwiki.defaultprefs.template.appearance | TRUE, FALSE | compile-time |
| jspwiki.defaultprefs.template.autosuggest | TRUE, FALSE | compile-time |
| jspwiki.defaultprefs.template.tabcompletion | TRUE, FALSE | compile-time |

Table 4.1 presents the configuration space model used for JSPWiki in the experiments. The model included 17 compile-time options with varying number of settings.

We had 12 options with 2 levels of settings, 3 options with 3 levels of settings, 1 option with 4 levels of setting and 1 option with 5 levels of settings.

Table 4.2 Configuration space model used for Apache HBase.

| Configuration Options | Settings | Type |
|---|---|---|
| hbase.master.infoserver.redirect | TRUE,FALSE | compile-time |
| hbase.regionserver.info.port.auto | TRUE,FALSE | compile-time |
| hfile.block.cache.policy | LRU, TinyLFU | compile-time |
| hbase.storescanner.parallel.seek.enable | TRUE,FALSE | compile-time |
| dfs.client.read.shortcircuit | TRUE,FALSE | compile-time |
| hbase.hstore.checksum.algorithm | NULL, CRC32, CRC32C | compile-time |
| hbase.regionserver.checksum.verify | TRUE,FALSE | compile-time |
| hbase.security.visibility.mutations.checkauths | TRUE,FALSE | compile-time |
| hbase.regionserver.handler.abort.on.error.percent | -1, 0, 0.5, 1 | compile-time |
| hbase.regionserver.region.split.policy | BusyRegionSplitPolicy, ConstantSizeRegionSplitPolicy, DisabledRegionSplitPolicy, DelimitedKeyPrefixRegionSplitPolicy, SteppingSplitPolicy | compile-time |

Similarly, Table 4.2 presents the configuration space model used for HBase in the experiments. The model involved 10 compile-time options with varying number of settings. We had 7 options with 2 levels of settings, 1 option with 3 levels of settings, 1 option with 4 levels of setting, and 1 option with 5 levels of settings.

### 4.1.4 Time Intervals

To perform the experiments, we also needed top determine a date interval, during which the proposed CIT-based daily process was carried out.

To this end, we basically examined the code repositories of our subject applications to determine a time frame during which the application was under development. This enabled us to determine a potential start date for the experiments.

Once this was done, the next question was how to determine the duration of the experiments (i.e., the number of days, during which the daily build process needed to be carried out). To this end, we opted to choose the duration with respect to our most sophisticated strategy, namely Strategy 4. In particular, for the configurations space model and the strategy parameters used for each subject application, we determined the maximum number of days required to run Strategy 4 to completion and used this number as the duration of the experiments. While doing so, we picked $k$ (Section 3.1.4) such that number of configurations required for testing matches that of standard covering arrays used by Strategy 1 (Section 3.1.1).

That is, we not only used the same time interval for each strategy, but also made sure that each strategy tested about the same number of configurations each day, so that we can compare the results obtained from different strategies.

For JSPWiki, we opted to use the time interval from 18-05-2020 to 29-05-2020, during which 3 commits were made (21st, 23rd, and 25th of May) (Foundation, 2020d). And, for HBase, we used the interval from 20-07-2020 to 28-07-2020, during which 7 commits were made (20th, 21st, 22nd, 23rd, 24th, 27th, and 28th of July) (Foundation, 2020b).

During the experiments, if a strategy ran to completion, we restarted the strategy from scratch and made sure that the strategy randomly selected its covering arrays to be tested.

## 4.2 Evaluation Framework

To evaluate the proposed approach, we used the following metrics:

- **test runs:** Total number of test cases executed.
- **failed test runs:** Total number of test cases failed.
- **passed test runs:** Total number of test cases passed.
- **skipped test runs:** Total number of test cases skipped; when the assumptions of a test case is not met by the underlying configuration the test case refused to run on the configuration, i.e., skipped the configuration.
- **distinct errors:** Total number of distinct errors observed. The errors observed during testing was obtained by parsing the error logs. The more distinct errors observed, the better the proposed approach is.
- **distinct error-test case pairs:** Total number of distinct error-test case pairs observed. For example, in Table 4.3, while we have 5 distinct error-test case pairs, we only have 3 distinct errors. The more distinct error-test case pairs observed, the better the proposed approach is.
- **structural code coverage:** Amount of instruction, line, branch, and method coverage obtained (Section 2.7).

Table 4.3 An example of test case and distinct error pair

| Configuration | Distinct Error 1 | Distinct Error 2 | Distinct Error 3 |
|---|---|---|---|
| cfg1 | TestCase-1 | TestCase-3 | - |
| cfg2 | - | TestCase-2 | - |
| cfg3 | TestCase-4 | - | TestCase-3 |

We used these metrics both on a daily basis and on a strategy basis. The former was interested in the values of these metrics after testing the selected covering array every day. The latter was, on the other hand, interested in the values of these metrics after carrying out a strategy during the selected time frame.

## 4.3 Operational Framework

Throughout the experiments, we used the *surefire* plugin (v3.0.0.M4) with the *maven* build system (Foundation, 2020e) to configure and build the SUDs as well as to run the test cases and collect the results. Furthermore, to obtain the structural code coverage statistics, we used the Jacoco plugin (v0.8.3) (Mountainminds GmbH Co. KG, 2020).

To compute the covering arrays, we used Jenny – a well-known covering array generator (Jenkins, 2020). Another frequently-used generator was ACTS (NIST, 2020). We, however, opted not to use this generator as it did not support the generation of randomized covering arrays – a feature which is quite curicial for CIT-daily.

All the experiments were carried out on Google Cloud by using computing engines running Ubuntu 16.04 on 1 vCPU and 3.75 GB memory per vCPU.

## 4.4 Data and Analysis

In the experiments, we used two experimental setups for Strategy 4; one where $t_1 = 2$ and $t_2 = 3$, and the other where $t_1 = 2$ and $t_2 = 4$. In the remainder of the document, these setups will be referred to as *first experimental setup* and *second experimental* *setup*, respectively. Furthermore, see Section 3.1.4 for more information about the parameters used in these setups.

Consequently, to be able to compare the results of different strategies, we carried out the remaining strategies with respect to these settings (Section 4.1.4. That is, for each setup we determined the number of days Strategy 4 needed for completion while testing the similar number of configuration options with standard 2-way covering arrays (i.e., the covering arrays used by Strategies 1, 2, and 3). It turned out for the first experimental setup where $t_1 = 2$ and $t_2 = 3$, the duration was 9 days for JSPWiki and 3 days for HBase. And, for the other setup, where $t_1 = 2$ and $t_2 = 4$, the duration was 12 days for JSPWiki and 9 days for HBase. We, therefore, ran the remaining strategies (Strategies 1, 2, and 3) for the aforementioned number of days for each experimental setup and subject application combination. When a strategy finished before the required number of days had expired, we re-ran the strategy from scratch.

With respect to the specified time intervals, Strategy 2 covers 99.95 and 96.41 percent of all the 3-tuples in first experimental setup for JSPWiki and HBase, respectively. And, for the second experimental setup, Strategy 2 covers 99.01 and 96.84 percent of all the 4-tuples for JSPWiki and HBase, respectively. Similarly, Strategy 3 covers 79.16 and 36.34 percent of the 3-tuples in the first experimental setup and 92.03 and 82.09 percent of the 4-tuples in the second experimental setup for JSPWiki and HBase, respectively.

We, furthermore, compared the results obtained from the CIT-daily processes to those obtained from the standard practice. In the standard practice, the default configuration of the SUD is used every day during the daily build process, unless test cases extensively change the configuration (e.g., runtime configuration) of the system. To mimic, this standard practice, ever day during the experiments, we downloaded the latest version of the SUD, built it with the default configuration (i.e., without performing any configuration), and ran the test cases on the default configuration. In the remainder of the document, this strategy is marked as "standard."

For JSPWiki, Tables 4.4-4.12 and Tables 4.14-4.22 present the results we obtained from different strategies for JSPWiki and HBase, respectively. In these tables, the columns depict the dates on which the experiments were carried out, test runs, test runs passed, test runs failed, test runs skipped, number of distinct errors, number of distinct error-test case pairs, and various structural code coverage statistics obtained, respectively. Also, the last rows in these table report the overall results obtained at the end of the strategy.

Furthermore, Tables 4.13 and 4.23 present the overall results obtained from various strategies under different experimental results for JSPWiki and HBase, respectively.

Table 4.4 JSPWiki: Results obtained from the standard daily build process where the default configuration of the SUD is used every day.

| repo date | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 19.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 20.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 21.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 22.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 23.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 24.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 25.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 26.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 27.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 28.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| 29.05.2020 | 1023 | 1023 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| Overall | 12276 | 12276 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |

Table 4.5 JSPWiki: Results obtained from Strategy 1 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 19.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 20.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 21.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 22.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 23.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 24.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 25.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 26.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| Overall | 24 | 220968 | 220104 | 864 | 4 | 4 | 9294 | 244 | 1418 | 40825 |

Table 4.6 JSPWiki: Results obtained from Strategy 1 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 19.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 20.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 21.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 22.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 23.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 24.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 25.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 26.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 27.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 28.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 29.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| Overall | 24 | 294624 | 293472 | 1152 | 4 | 4 | 9294 | 244 | 1418 | 40825 |

Table 4.7 JSPWiki: Results obtained from Strategy 2 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 22 | 22506 | 22418 | 88 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 19.05.2020 | 23 | 23529 | 23437 | 92 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 20.05.2020 | 23 | 23529 | 23437 | 92 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 21.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 22.05.2020 | 25 | 25575 | 25425 | 150 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 23.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 24.05.2020 | 25 | 25575 | 25425 | 150 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 25.05.2020 | 22 | 22506 | 22374 | 132 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 26.05.2020 | 23 | 23529 | 23437 | 92 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| Overall | 187 | 213807 | 212735 | 1072 | 6 | 6 | 9305 | 245 | 1418 | 40849 |

Table 4.8 JSPWiki: Results obtained from Strategy 2 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 22 | 22506 | 22352 | 154 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 19.05.2020 | 25 | 25575 | 25400 | 175 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 20.05.2020 | 25 | 25575 | 25400 | 175 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 21.05.2020 | 23 | 23529 | 23368 | 161 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 22.05.2020 | 23 | 23529 | 23368 | 161 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 23.05.2020 | 23 | 23529 | 23368 | 161 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 24.05.2020 | 24 | 24552 | 24384 | 168 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 25.05.2020 | 23 | 23529 | 23368 | 161 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 26.05.2020 | 25 | 25575 | 25400 | 175 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 27.05.2020 | 24 | 24552 | 24384 | 168 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 28.05.2020 | 23 | 23529 | 23368 | 161 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| 29.05.2020 | 25 | 25575 | 25400 | 175 | 7 | 7 | 7941 | 172 | 1074 | 36491 |
| Overall | 259 | 291555 | 289560 | 1995 | 7 | 7 | 9417 | 245 | 1418 | 40894 |

Table 4.9 JSPWiki: Results obtained from Strategy 3 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 22 | 22506 | 22418 | 88 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 19.05.2020 | 23 | 23529 | 23437 | 92 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 20.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 21.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 22.05.2020 | 24 | 24552 | 24408 | 144 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 23.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 24.05.2020 | 25 | 25575 | 25425 | 150 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 25.05.2020 | 24 | 24552 | 24408 | 144 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 26.05.2020 | 23 | 23529 | 23437 | 92 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| Overall | 197 | 215853 | 214771 | 1082 | 6 | 6 | 9305 | 245 | 1418 | 40849 |

Table 4.10 JSPWiki: Results obtained from Strategy 3 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 22 | 22506 | 22418 | 88 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 19.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 20.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 21.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 22.05.2020 | 24 | 24552 | 24408 | 144 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 23.05.2020 | 22 | 22506 | 22374 | 132 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 24.05.2020 | 25 | 25575 | 25425 | 150 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 25.05.2020 | 24 | 24552 | 24408 | 144 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 26.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 27.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 28.05.2020 | 24 | 24552 | 24456 | 96 | 4 | 4 | 7976 | 172 | 1075 | 36616 |
| 29.05.2020 | 25 | 25575 | 25425 | 150 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| Overall | 244 | 288486 | 286934 | 1552 | 6 | 6 | 9305 | 245 | 1418 | 40849 |

Table 4.11 JSPWiki: Results obtained from Strategy 4 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 19.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 20.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 21.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 22.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 23.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 24.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 25.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| 26.05.2020 | 23 | 23529 | 23391 | 138 | 6 | 6 | 7937 | 172 | 1073 | 36491 |
| Overall | 184 | 211761 | 210519 | 1242 | 6 | 6 | 9305 | 245 | 1418 | 40849 |

Table 4.12 JSPWiki: Results obtained from Strategy 4 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 18.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 19.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 20.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 21.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 22.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 23.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 24.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 25.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 26.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 27.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 28.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| 29.05.2020 | 23 | 23529 | 23276 | 253 | 11 | 11 | 7937 | 172 | 1074 | 36483 |
| Overall | 239 | 282348 | 279312 | 3036 | 11 | 11 | 9268 | 246 | 1417 | 40713 |

Table 4.13 JSPWiki: Overall results obtained from various strategies under different experimental setups.

| setup | strategy | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Standard | Standard | 1 | 12276 | 12276 | 0 | 0 | 0 | 0 | 9293 | 243 | 1418 | 40811 |
| Setup 1 | Strategy 1 | 24 | 220968 | 220104 | 864 | 0 | 4 | 4 | 9294 | 244 | 1418 | 40825 |
| Setup 1 | Strategy 2 | 187 | 213807 | 212735 | 1072 | 0 | 6 | 6 | 9305 | 245 | 1418 | 40849 |
| Setup 1 | Strategy 3 | 197 | 215853 | 214771 | 1082 | 0 | 6 | 6 | 9305 | 245 | 1418 | 40849 |
| Setup 1 | Strategy 4 | 184 | 211761 | 210519 | 1242 | 0 | 6 | 6 | 9305 | 245 | 1418 | 40849 |
| Setup 2 | Strategy 1 | 24 | 294624 | 293472 | 1152 | 0 | 4 | 4 | 9294 | 244 | 1418 | 40825 |
| Setup 2 | Strategy 2 | 259 | 291555 | 289560 | 1995 | 0 | 7 | 7 | 9417 | 245 | 1418 | 40894 |
| Setup 2 | Strategy 3 | 244 | 288486 | 286934 | 1552 | 0 | 6 | 6 | 9305 | 245 | 1418 | 40849 |
| Setup 2 | Strategy 4 | 239 | 282348 | 279312 | 3036 | 0 | 11 | 11 | 9268 | 246 | 1417 | 40713 |

Table 4.14 HBase: Results obtained from the standard daily build process where the default configuration of the SUD is used every day.

| repo date | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68676 | 14886 | 11893 | 341665 |
| 21.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68755 | 14930 | 11906 | 342020 |
| 22.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68701 | 14893 | 11896 | 341785 |
| 23.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68684 | 14905 | 11888 | 341734 |
| 24.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68752 | 14920 | 11907 | 342027 |
| 25.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68813 | 14949 | 11913 | 342237 |
| 26.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68634 | 14899 | 11875 | 341517 |
| 27.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68669 | 14884 | 11888 | 341697 |
| 28.07.2020 | 2073 | 2056 | 1 | 16 | 1 | 1 | 68632 | 14901 | 11873 | 341514 |
| Overall | 18657 | 18504 | 9 | 144 | 1 | 1 | 47458 | 8987 | 8861 | 243647 |

Table 4.15 HBase: Results obtained from Strategy 1 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53182 | 10043 | 8773 | 268236 |
| 21.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53194 | 10051 | 8774 | 268258 |
| 22.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53196 | 10051 | 8774 | 268300 |
| Overall | 22 | 69036 | 68046 | 198 | 792 | 1 | 1 | 47496 | 8993 | 8847 | 243745 |

Table 4.16 HBase: Results obtained from Strategy 1 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53182 | 10043 | 8773 | 268236 |
| 21.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53194 | 10051 | 8774 | 268258 |
| 22.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53196 | 10051 | 8774 | 268300 |
| 23.07.2020 | 22 | 23012 | 22660 | 88 | 264 | 2 | 2 | 53198 | 10054 | 8773 | 268306 |
| 24.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53192 | 10043 | 8771 | 268263 |
| 25.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53213 | 10056 | 8775 | 268364 |
| 26.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53213 | 10056 | 8775 | 268364 |
| 27.07.2020 | 22 | 23012 | 22660 | 88 | 264 | 2 | 2 | 53198 | 10054 | 8773 | 268306 |
| 28.07.2020 | 22 | 23012 | 22682 | 66 | 264 | 1 | 1 | 53194 | 10051 | 8774 | 268258 |
| Overall | 22 | 207108 | 204094 | 638 | 2376 | 2 | 2 | 53209 | 10066 | 8776 | 268371 |

Table 4.17 HBase: Results obtained from Strategy 2 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 22 | 46442 | 42548 | 3542 | 352 | 2 | 2 | 53234 | 10040 | 8795 | 268377 |
| 21.07.2020 | 21 | 46221 | 42504 | 3381 | 336 | 2 | 2 | 53219 | 10042 | 8790 | 268302 |
| 22.07.2020 | 22 | 46398 | 42504 | 3542 | 352 | 2 | 2 | 53139 | 10032 | 8769 | 268020 |
| Overall | 59 | 139061 | 127556 | 10465 | 688 | 2 | 2 | 53215 | 10040 | 8793 | 268392 |

Table 4.18 HBase: Results obtained from Strategy 2 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 21 | 44289 | 40530 | 3423 | 336 | 4 | 4 | 53274 | 10057 | 8822 | 268544 |
| 21.07.2020 | 21 | 44310 | 40572 | 3402 | 336 | 3 | 3 | 53168 | 10058 | 8770 | 268171 |
| 22.07.2020 | 22 | 46398 | 42504 | 3542 | 352 | 2 | 2 | 53139 | 10032 | 8769 | 268020 |
| 23.07.2020 | 22 | 46420 | 42482 | 3586 | 352 | 4 | 4 | 53266 | 10062 | 8808 | 268514 |
| 24.07.2020 | 22 | 23012 | 22638 | 110 | 264 | 3 | 3 | 53238 | 10053 | 8790 | 268438 |
| 25.07.2020 | 22 | 23012 | 22660 | 88 | 264 | 2 | 2 | 53016 | 10039 | 8757 | 267453 |
| 26.07.2020 | 21 | 44289 | 40530 | 3423 | 336 | 4 | 4 | 53274 | 10057 | 8822 | 268544 |
| 27.07.2020 | 22 | 23012 | 22638 | 110 | 264 | 3 | 3 | 53157 | 10046 | 8770 | 268102 |
| 28.07.2020 | 21 | 44310 | 40572 | 3402 | 336 | 3 | 3 | 53168 | 10058 | 8770 | 268171 |
| Overall | 178 | 339052 | 315126 | 21086 | 2840 | 4 | 4 | 53333 | 10089 | 8882 | 268985 |

Table 4.19 HBase: Results obtained from Strategy 3 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 22 | 46442 | 42504 | 3586 | 352 | 4 | 4 | 53150 | 10031 | 8774 | 268032 |
| 21.07.2020 | 21 | 44289 | 40572 | 3381 | 336 | 2 | 2 | 53140 | 10042 | 8771 | 268013 |
| 22.07.2020 | 22 | 23012 | 22660 | 88 | 264 | 2 | 2 | 52334 | 10049 | 8790 | 268407 |
| Overall | 61 | 113743 | 105736 | 7055 | 952 | 4 | 4 | 53455 | 10056 | 8792 | 268245 |

Table 4.20 HBase: Results obtained from Strategy 3 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 22 | 46442 | 42548 | 3542 | 352 | 2 | 2 | 53234 | 10040 | 8795 | 268377 |
| 21.07.2020 | 21 | 44289 | 40572 | 3381 | 336 | 2 | 2 | 53219 | 10042 | 8790 | 268302 |
| 22.07.2020 | 22 | 46398 | 42504 | 3542 | 352 | 2 | 2 | 53139 | 10032 | 8769 | 268020 |
| 23.07.2020 | 22 | 46398 | 42482 | 3564 | 352 | 3 | 3 | 53239 | 10050 | 8790 | 268421 |
| 24.07.2020 | 22 | 23012 | 22638 | 110 | 264 | 3 | 3 | 53157 | 10046 | 8770 | 268102 |
| 25.07.2020 | 21 | 44310 | 40551 | 3423 | 336 | 4 | 4 | 53319 | 10051 | 8803 | 268755 |
| 26.07.2020 | 22 | 23012 | 22660 | 88 | 264 | 2 | 2 | 53016 | 10039 | 8757 | 267453 |
| 27.07.2020 | 21 | 44289 | 40530 | 3423 | 336 | 4 | 4 | 53274 | 10057 | 8822 | 268544 |
| 28.07.2020 | 22 | 46420 | 42482 | 3586 | 352 | 4 | 4 | 53266 | 10062 | 8808 | 268514 |
| Overall | 187 | 364570 | 336967 | 24659 | 2944 | 4 | 4 | 53393 | 10067 | 8824 | 268773 |

Table 4.21 HBase: Results obtained from Strategy 4 for the first experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 21 | 22029 | 21609 | 168 | 252 | 6 | 6 | 53200 | 10022 | 8775 | 268228 |
| 21.07.2020 | 22 | 23012 | 22572 | 176 | 264 | 6 | 6 | 53194 | 10022 | 8778 | 268228 |
| 22.07.2020 | 22 | 23012 | 22572 | 176 | 264 | 6 | 6 | 53180 | 10023 | 8770 | 268180 |
| Overall | 60 | 68053 | 66753 | 520 | 780 | 6 | 6 | 532196 | 10017 | 8778 | 268220 |

Table 4.22 HBase: Results obtained from Strategy 4 for the second experimental setup.

| repo date | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20.07.2020 | 21 | 44289 | 40467 | 3486 | 336 | 7 | 7 | 53054 | 10024 | 8758 | 267584 |
| 21.07.2020 | 21 | 44289 | 40467 | 3486 | 336 | 7 | 7 | 53039 | 10014 | 8758 | 267532 |
| 22.07.2020 | 22 | 23012 | 22550 | 198 | 264 | 7 | 7 | 53041 | 10014 | 8758 | 267541 |
| 23.07.2020 | 22 | 46420 | 42438 | 3630 | 352 | 6 | 6 | 53184 | 10017 | 8770 | 268191 |
| 24.07.2020 | 21 | 44310 | 40551 | 3423 | 336 | 4 | 4 | 53319 | 10051 | 8803 | 268755 |
| 25.07.2020 | 22 | 23012 | 22572 | 176 | 264 | 6 | 6 | 53196 | 10090 | 8770 | 268250 |
| 26.07.2020 | 22 | 23012 | 22572 | 176 | 264 | 6 | 6 | 53193 | 10024 | 8770 | 268227 |
| 27.07.2020 | 22 | 23012 | 22572 | 176 | 264 | 6 | 6 | 53180 | 10023 | 8770 | 268180 |
| 28.07.2020 | 22 | 23012 | 22572 | 176 | 264 | 6 | 6 | 53180 | 10023 | 8770 | 268180 |
| Overall | 177 | 294368 | 276761 | 14927 | 2680 | 7 | 7 | 53345 | 10090 | 8813 | 268984 |

Table 4.23 HBase: Overall results obtained from various strategies under different experimental setups.

| setup | strategy | distinct cfgs tested | test runs | tests passed | tests failed | tests skipped | distinct errs. | distinct err-test pairs | line cov. | branch cov. | method cov. | instruction cov. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Standard | Standard | 1 | 18657 | 18504 | 9 | 144 | 1 | 1 | 47458 | 8987 | 8861 | 243647 |
| Setup 1 | Strategy 1 | 22 | 69036 | 68046 | 198 | 792 | 1 | 1 | 47496 | 8993 | 8847 | 243745 |
| Setup 1 | Strategy 2 | 59 | 139061 | 127556 | 10817 | 688 | 2 | 2 | 53215 | 10040 | 8793 | 268392 |
| Setup 1 | Strategy 3 | 61 | 113743 | 105736 | 7055 | 952 | 4 | 4 | 53455 | 10056 | 8792 | 268245 |
| Setup 1 | Strategy 4 | 60 | 68053 | 66753 | 520 | 780 | 6 | 6 | 532196 | 10017 | 8778 | 268220 |
| Setup 2 | Strategy 1 | 22 | 207108 | 204094 | 638 | 2376 | 2 | 2 | 53209 | 10066 | 8776 | 268371 |
| Setup 2 | Strategy 2 | 178 | 339052 | 315126 | 21086 | 2840 | 4 | 4 | 53333 | 10089 | 8882 | 268985 |
| Setup 2 | Strategy 3 | 187 | 364570 | 336967 | 24659 | 2944 | 4 | 4 | 53393 | 10067 | 8824 | 268773 |
| Setup 2 | Strategy 4 | 177 | 294368 | 276761 | 14927 | 2680 | 7 | 7 | 53345 | 10090 | 8813 | 268984 |

We first observed that among all the strategies that can be used the worst performing one was the standard strategy, where the default configuration of the SUD is used every day to test the system. This strategy while revealing no failures for JSPWiki (Table 4.13), revealed only one distinct error for HBase (Table 4.23).

We next observed that Strategy 1, where the same $t$-way covering array used everyday during the daily build process for testing, performed profoundly better than the standard strategy. Strategy 1 revealed 4 distinct errors under each experimental setups for JSPWiki (Table 4.13). For HBase, it revealed 1 and 2 distinct errors for the first and second experimental setups, respectively (Table 4.23).

We then observed that Strategy 2 and 3, where the covering array used every day changed, had similar performances, which were profoundly better than Strategy 1. For example, Strategy 3 revealed 6 distinct errors under each experimental setup for JSPWiki and 4 distinct errors under each experimental setup for HBase. We finally observed that among all the the strategies we experimented with the best strategy was Strategy 4. For JSPWiki, Strategy 4 revealed 6 and 11 distinct errors for the first and second experimental setups, respectively. And, for HBase, it revealed 6 and 7 distinct errors for the first and second experimental setups, respectively. Note further that all these achievements were obtained by testing the same or similar number configurations, i.e., at the same or similar testing costs, compared to Strategy 1, 2, and 3.

Last but not least, we observed that as $t_2$ increased we tented to have better results, especially for Strategy 4. For example, when $t_2$ was increased from 3 to 4 for JSPWiki, the number of distinct errors revealed by Strategy 4 was increased from 6 to 11.

### 4.4.1 Discussion

As represented results within tables, we expected that coverage metrics would be consistent. This inconsistency can be caused by selecting tool in order to calculate coverage metric or having random covering array. We cannot strictly track which configuration is related to where in the code base because of having randomly generated covering array.

Another point is that we have also expected that distinct error count is different than distinct error - test case pairs but we cannot observe this case. Each error is related to one specific test case in selected subject applications. Last but not least, the daily build processes are typically expected to run throughout the night (in 6 to 8 hours) while the developers are not in their offices. In the experiments, we tested around 22 configurations per day for JSPWiki, each of which took about 2 hours, and around 24 configurations per day for HBase, each of which took about 2.5 hours. That is, when a single computer were used for testing, the process would have taken more than a day to execute. Note, however, that multiple computers can always be used to test the configurations included in a covering array in parallel, such that the entire process can be carried out during a single night.

# 5. THREATS TO VALIDITY

We have identified various threats to validity for these experiments.

First threat is that we conducted experiments on two subject applications. It may affect generality of our results.

Second threat can be that we created configuration space model manually. So chosen configuration option may affect the results. We cannot ensure that the option in the model is the best choice for selected subject application.

Third threat is that all test cases of chosen applications is implemented by their developers. It is impossible to examine the content and quality of all test cases.

Fourth threat is that we selected Jacoco plugin for collecting coverage result. Even though it is commonly used for maven project, accuracy of the plugin and whether another plugin gives approximate results are questionable points.

However, we believe that CIT-daily is effective and successful to reveal more failures than applying standard daily build process without any combinatorial interaction testing approach. Also, experiments in this thesis represent accurate and consistent results.

# 6. RELATED WORK

This section has represented related work of what has been done about the concepts that are used in the thesis so far. We have mainly focused on how to be determined configuration space model, how be generated covering array, how to be used seeds and also what kind of studies on testing process in daily build process are.

## 6.1 Configuration Space Model

To determine configuration space model is baseline for combinatorial interaction testing. Before working with CIT approach, developers or testers are expected to decide on this model. To reduce this effort on their side and strength the accuracy of the determined model, a study is conducted by Charles Song and their colleagues. Song, Porter & Foster (2013) have proposed an algorithm called as iTree. It aims to achieve high coverage by using a tiny subset of whole configuration space. The process of algorithm continues iterative with the help of machine learning techniques. It starts testing a subject application with low strength covering array produced in CIT approach then it gathers coverage results. By using these results and machine learning techniques, it observes new interactions that can be candidate for next covering array in order to increase totally coverage ratio then it repeats these steps. Finally, the proposed algorithm gives a set of configuration space that achieves high coverage and set of configurations containing high strength interaction (Song et al., 2013) (Yilmaz et al., 2013).

## 6.2 Covering Array

Covering array is set of configurations and used for testing subject application in combinatorial interaction testing approach. Three different types of it is mentioned in Section 2.5 and constructing covering array can be categorized in two main strategies that are called as one parameter at a time (OPAT) and one test at a time (OTAT) strategy (Grindal et al., 2005) (Grindal, Offutt & Andler, 2005) (Ahmed & Zamli, 2011). So, to construct covering array is NP-Hard problem so it requires great effort to find common construction method (Lei & Tai, 1998) (Hartman & Raskin, 2004). Because of this situation, there are different tools and methods to solve the case thus we have easily founded various studies in the literature related to this concept.

We have gathered reviewed studies under two main part based on strategies mentioned above.

First part is related to algorithms developed based on OPAT strategy. A logic of the strategy is to construct covering array by starting with smallest number of options and check its coverage in predetermined test suite. Per iteration, the covering array extends with adding one option in horizontal (Nasser, Sariera, Alsewari & Zamli, 2015). The most used construction algorithms developed depending on this strategy is In-Parameter-OrderGeneral (IPOG) (Lei, Kacker, Kuhn, Okun & Lawrence, 2007) and IPO-s (Calvagna & Gargantini, 2009) IPOG-F and IPOG-D (Lei, Kacker, Kuhn, Okun & Lawrence, 2008) and IPAD2 (Forbes, Lawrence, Lei, Kacker & Kuhn, 2008) Also, Klaib et al. (2010) developed different approach called as a tree based generation for only be used in pairwise testing (Klaib, Muthuraman, Ahmad & Sidek, 2010).

Another part is related to algorithms developed based on OTAT strategy. The strategy takes into account only one row of the covering array as test case per iteration and checks coverage ratio of the selected row. The process iterates till all combinations are covered (Nasser et al., 2015). The well known algorithms for the strategy can be listed as The Automatic Efficient Test Generator (AETG) (Cohen et al., 1997), mAETG (Cohen, 2004) (Cohen, 2004) , Pairwise Independent Combinatorial Testing (PICT) (Czerwonka, 2008), Deterministic Density Algorithm (DDA) (Bryce & Colbourn, 2007) (Bryce & Colbourn, 2009), Classification-Tree Editor eXtended Logics (CTE-XL) (Lehmann & Wegener, 2000) (Yu, Ng & Chan, 2003), Test Vector Generator (TVG) (Tung & Aldiwan, 2000) Jenny (2010) (Jenkins, 2020) and WHITCH (Hartman, Klinger & Raskin, 2010).

On the other hand, all developed algorithms or strategies have to cope with optimization problem related to size of covering array. To solve this optimization problem, the more popular methods that are greedy (Cohen et al., 1997) (Czerwonka, 2008), random search-based (Nie, Xu & Leung, 2009) (Schroeder, Bolaki & Gopu, 2004), heuristic search-based (Ghazi & Ahmed, 2003), mathematical methods (Grindal, Offutt & Mellin, 2006) are placed in developed algorithms. While algorithms based on OTAT strategy use heuristic search based methods, algorithms based on OPAT strategy use greedy methods (Ahmed & Zamli, 2011).

These optimization methods is good at high interaction strength. Besides them, there are also different developed algorithm to give good result for small interaction strength. These can be listed as simulated Annealing (SA) (Stardom, 2001),Genetic Algorithm (GA), Ant Colony Algorithm (ACA) (Shiba, Tsuchiya & Kikuno, 2004), Tabu Search (TS) (Nurmela, 2004), Particle Swarm Optimization (Ahmed & Zamli, 2010).

## 6.3 Seeds

Seeding is one of approaches in the literature to construct covering array. It has a set of configurations from t-1 way covering array as initial step then generate new covering array containing both all t-1 way interactions in seed and t-way interactions that are not equal with interactions in seed. Fouché et al. (2009) and Fouché, Cohen & Porter (2007) use this approach to construct covering array schedules. By reusing tested configurations from covering array produced in lower strength as a seed, it is aimed to increase strength for new covering array. Tai & Lei (2002) uses this approach to generate new covering array but its goal is not increasing strength. It produces covering array in same t-way with t-way of seed. Yilmaz (2012) constructs test aware covering array to reduce total test run by sharing configurations in seed. Cohen et al. (2008) also measures efficiency of this mechanism in their study.

On the other side, we have used seeding mechanism with opposite direction in our work. Generally the literature contains that seed is created from configurations of t-1 way covering array then it is used in generating t-way covering array. However, we have implemented that a seed is created from set of configurations of t-way covering array and we have used it to generate t-1 way covering array. By this structure, we have made sure that all configurations in t-way covering array are covered in t-1 way covering array within certain days.

## 6.4 Daily Build

There are huge number of studies about daily build. It is part of continuous integration (CI) in agile driven development. On the other hand, there are very limited resources about running different testing approach in daily build process. Memon et al. (2003) introduces running smoke tests in daily build with the help of their automated framework called as DART. Smoke test is for validating basic functionality not exhaustive testing so the framework has focused on graphical user interface (GUI) retesting. McConnell (1996) gives detail information about advantage of running smoke testing in daily.

There are no other sources than two studies. For this case, we have aimed to improve the concept that combines combinatorial interaction testing approach with daily build process rather than only focusing on smoke testing. So, any testing stage like smoke, regression or integration can be implemented or extended based on proposed methods.

# 7. CONCLUDING REMARKS

Daily build process in agile development environment has predefined configurations to test subject application. These same configurations are used for the repository that is changed by commit. In this case, these configurations only cover specific test cases and remaining test cases may not be evaluated. It causes to have same failures in repository even if it has various hidden failures. For this situation, we have focused on rather than using same configuration on changing repository, it is able to reveal more failures by using different configurations on the repository.

To reach this goal, we developed an automated framework combining daily build process that contains stages configure, build and test repository in respectively for each day and combinatorial interaction testing approach with using different testing strategies. For running framework correctly, we have to determine configuration space model that is about whether configuration option is included or not and specify date interval for handling the repository. After initialization part, at each iteration in the framework, covering arrays is generated with respect to developed testing strategies, then last version of selected subject application is downloaded, configured based on newly generated configuration in covering arrays, built and tested. After completed work on the repository, all test and coverage results are gathered.

To observe real results obtained from the framework, we conducted two different experiments. Open source Apache JSPWiki and Apache HBase projects was selected to use in the experiments as subject applications. We firstly run general approach on the repository and got the number of total and distinct and failures, passing test cases and coverage information. We then rerun the repository with respect to developed testing strategies. We observed that the CIT-daily reveals more remarkable failures than general approach. Even if covering test cases are same with general approach, with the help of different configuration, the number of total test passed and failed changes the version of repository from one day to another.

Another remarkable observation is related to effective strategy among other developed strategies. It is obvious that each developed testing strategies is better than standard daily build process. However, the Strategy 4 is the best in case of detecting error in the repository earlier.

As a summary, the results of all experiments strongly support that it is more effective to use combinatorial interaction based daily build process than to use standard daily build process.

# 8. FUTURE WORK

After getting experiment results, we consider that there is more studies to be done in this research area.

One issue we are planning to focus is about developing different testing strategies to support our approach.

We will also enhance the framework to support different covering array generation tools like ACTS, PICT because in this research we have only used Jenny tool.

Another issue is to carry out the experiments by using more complex configurations space with larger subject applications.

Besides this, in the thesis we cannot provide clear results about coverage information because of randomness. If it can be tracked which configuration has impact on which code block on the software development with the help of changing on coverage, it is easy to figure out most important configuration interaction for chosen software.

Finally, we will use different techniques to determine configuration option of the subject application rather than doing manually to support automation of the developed framework.

# BIBLIOGRAPHY

Ahmed, B. & Zamli, K. (2011). A review of covering arrays and their application to software testing. *Journal of Computer Science*, *7*, 1375–1385.

Ahmed, B. S., Abdulsamad, T. S., & Potrus, M. Y. (2015). Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm. *Information and Software Technology*, *66*, 13–29.

Ahmed, B. S., Gargantini, A., Zamli, K. Z., Yilmaz, C., Bures, M., & Szeles, M. (2019). Code-aware combinatorial interaction testing. *IET Software*, *13*(6), 600–609.

Ahmed, B. S. & Zamli, K. Z. (2010). Pstg: a t-way strategy adopting particle swarm optimization. In *2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, (pp. 1–5). IEEE.

Bryce, R. C. & Colbourn, C. J. (2006). Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, *48*(10), 960–970.

Bryce, R. C. & Colbourn, C. J. (2007). The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, *17*(3), 159–182.

Bryce, R. C. & Colbourn, C. J. (2009). A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability*, *19*(1), 37–53.

Calvagna, A. & Gargantini, A. (2009). Ipo-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays. In *2009 International conference on software testing, verification, and validation workshops*, (pp. 10–18). IEEE.

Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, *23*(7), 437–444.

Cohen, M. B. (2004). Designing test suites for software interactions testing. Technical report, AUCKLAND UNIV (NEW ZEALAND).

Cohen, M. B., Colbourn, C. J., & Ling, A. C. (2008). Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*, *308*(13), 2709–2722.

Czerwonka, J. (2008). Pairwise testing in the real world: Practical extensions to test-case scenarios. *Microsoft Corporation, Software Testing Technical Articles.*

Forbes, M., Lawrence, J., Lei, Y., Kacker, R. N., & Kuhn, D. R. (2008). Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, *113*(5), 287.

Fouché, S., Cohen, M. B., & Porter, A. (2009). Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, (pp. 177–188).

Fouché, S., Cohen, M., & Porter, A. (2007). Towards incremental adaptive covering arrays. (pp. 557–560).

Foundation, A. S. (2007 (accessed August, 2020)a). *Apache HBase.*

Foundation, A. S. (2007 (accessed August, 2020)b). *Apache HBase Repository.*

Foundation, A. S. (2013 (accessed August, 2020)c). *Apache JSPWiki.*

Foundation, A. S. (2013 (accessed August, 2020)d). *Apache JSPWiki Repository.*

Foundation, A. S. (2020 (accessed August, 2020)e). *Maven Surefire Plugin.*

Ghazi, S. A. & Ahmed, M. A. (2003). Pair-wise test coverage using genetic algorithms. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, volume 2, (pp. 1420–1424). IEEE.

Grindal, M., Offutt, J., & Andler, S. F. (2005). Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, *15*(3), 167–199.

Grindal, M., Offutt, J., & Mellin, J. (2006). *Handling constraints in the input space when using combination strategies for software testing.* Institutionen för kommunikation och information.

Grinwald, R., Harel, E., Orgad, M., Ur, S., & Ziv, A. (1998). User defined coverage-a tool supported methodology for design verification. In *Proceedings 1998 Design and Automation Conference. 35th DAC.(Cat. No. 98CH36175)*, (pp. 158–163). IEEE.

Hartman, A., Klinger, T., & Raskin, L. (2010). Ibm intelligent test case handler. *Discrete Mathematics*, *284*(1), 149–156.

Hartman, A. & Raskin, L. (2004). Problems and algorithms for covering arrays. *Discrete Mathematics*, *284*(1-3), 149–156.

Jenkins, B. (2005 (accessed August, 2020)). *jenny: a pairwise testing tool.*

Karlsson, E.-A., Andersson, L.-G., & Leion, P. (2000). Daily build and feature development in large distributed projects. (pp. 649–658).

Klaib, M. F., Muthuraman, S., Ahmad, N., & Sidek, R. (2010). Tree based test case generation and cost calculation strategy for uniform parametric pairwise testing. *Journal of Computer Science*, *6*(5), 542.

Kobayashi, N., Tsuchiya, T., & Kikuno, T. (2002). A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters*, *81*(2), 85–91.

Kuhn, R., Kacker, R., Lei, Y., & Hunter, J. (2009). Combinatorial software testing. *Computer*, *42*(8), 94–96.

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

Kuliamin, V. V. & Petukhov, A. (2011). A survey of methods for constructing covering arrays. *Programming and Computer Software*, *37*(3), 121.

Lehmann, E. & Wegener, J. (2000). Test case design by means of the cte xl. In *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark.*

Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., & Lawrence, J. (2007). Ipog: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, (pp. 549–556). IEEE.

Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., & Lawrence, J. (2008). Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, *18*(3), 125–148.

Lei, Y. & Tai, K.-C. (1998). In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, (pp. 254–261). IEEE.

McConnell, S. (1996). Daily build and smoke test. *IEEE software*, *13*(4), 144.

Memon, A., Banerjee, I., Hashmi, N., & Nagarajan, A. (2003). Dart: a framework for regression testing" nightly/daily builds" of gui applications. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, (pp. 410–419). IEEE.

Memon, A., Nagarajan, A., & Xie, Q. (2005). Automating regression testing for evolving gui software. *J. Softw. Maintenance Res. Pract.*, *17*, 27–64.

Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D., & Natarajan, B. (2004). Skoll: Distributed continuous quality assurance. In *Proceedings. 26th International Conference on Software Engineering*, (pp. 459–468). IEEE.

Memon, A. M. & Qing Xie (2004). Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, (pp. 8–17).

Mohamed, N., Sulaiman, R. F. R., & Endut, W. R. W. (2013). The use of cyclomatic complexity metrics in programming performance's assessment. *Procedia-Social and Behavioral Sciences*, *90*, 497–503.

Mountainminds GmbH  Co. KG, M. (2009 (accessed August, 2020)). *Jacoco Coverage Counters.*

Nasser, A. B., Sariera, Y. A., Alsewari, A. A., & Zamli, K. Z. (2015). Assessing optimization based strategies for t-way test suite generation: the case for flower-based strategy. In *2015 IEEE international conference on control system, computing and engineering (ICCSCE)*, (pp. 150–155). IEEE.

Nie, C. & Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, *43*(2), 1–29.

Nie, C., Xu, B., & Leung, H. (2009). Using computational search to generate 2-way covering array. In *Proceedings of the International Symposium on Search Based Software Engineering, Fast Abstract.*

Nie, C., Xu, B., Shi, L., & Wang, Z. (2006). A new heuristic for test suite generation for pair-wise testing. In *SEKE*, (pp. 517–521).

NIST (2009 (accessed August, 2020)). *User Guide for ACTS.*

Nurmela, K. J. (2004). Upper bounds for covering arrays by tabu search. *Discrete applied mathematics*, *138*(1-2), 143–152.

Ramirez, A. J. & Cheng, B. H. (2010). Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, (pp. 49–58).

Schmidt, V. & Kruse, P. M. (2020). Test design with the classification tree method in presence of variants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 487–490). IEEE.

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

Shahid, D. M., Ibrahim, S., & Mahrin, M. (2011). A study on test coverage in software testing.

Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, *3*, 30 – 36.

Shiba, T., Tsuchiya, T., & Kikuno, T. (2004). Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual*

*International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, (pp. 72–77). IEEE.

Simos, D. E., Zivanovic, J., & Leithner, M. (2019). Automated combinatorial testing for detecting sql vulnerabilities in web applications. *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, 55–61.

Song, C., Porter, A., & Foster, J. S. (2013). itree: efficiently discovering high-coverage configurations using interaction trees. *IEEE Transactions on Software Engineering*, *40*(3), 251–265.

Stardom, J. (2001). *Metaheuristics and the search for covering and packing arrays.* Simon Fraser University Burnaby.

Tai, K.-C. & Lei, Y. (2002). A test generation strategy for pairwise testing. *IEEE transactions on software Engineering*, *28*(1), 109–111.

Tung, Y.-W. & Aldiwan, W. S. (2000). Automating test case generation for the new generation mission software system. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484)*, volume 1, (pp. 431–437). IEEE.

Watson, A. H., Wallace, D. R., & McCabe, T. J. (1996). *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of . . . .

Yilmaz, C. (2012). Test case-aware combinatorial interaction testing. *IEEE Transactions on Software Engineering*, *39*(5), 684–706.

Yilmaz, C., Fouche, S., Cohen, M. B., Porter, A., Demiroz, G., & Koc, U. (2013). Moving forward with combinatorial interaction testing. *Computer*, *47*(2), 37–45.

Yu, L., Lei, Y., Kacker, R. N., & Kuhn, D. R. (2013). Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, (pp. 370–375). IEEE.

Yu, Y., Ng, S. P., & Chan, E. Y. (2003). Generating, selecting and prioritizing test cases from specifications with tool support. In *Third International Conference on Quality Software, 2003. Proceedings.*, (pp. 83–90). IEEE.

Zimmerer, P. (2004). Combinatorial testing experiences, tools, and solutions.