Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Avocado framework plugin for generating combinatorial interaction tests

Jan Richter

Supervisor: Dr. BestounS. Ahmed Al-Beywanee, PH.D.
May 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Richter Jan**	Personal ID number:	**420069**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Avocado framework plugin for generating combinatorial interaction tests

Master’s thesis title in Czech:

Avocado framework plugin for generating combinatorial interaction tests

Guidelines:

The thesis goal is to design and develop a plugin for the Avocado testing framework, which will be generating combinatorial interaction tests. The plugin will be developed in Python and with be compatible with Avocado framework plugin system. The plugin accepts the combinatorial problem definition, composing of input parameters. The goal is to allow the plugin to accept the largest amount of these input parameters. Because of the nature of the problem, a particular limit cannot be defined in advance, but a minimal number of parameters is 200. The plugin has to handle constraints between parameters in the problem definition and parameter interaction strength up to 6. Test the created plugin with a suitable system of unit tests.

Bibliography / sources:

Kuhn, D. R., Kacker, R. N., & Lei, Y. (2013). Introduction to combinatorial testing. CRC press.
Kuhn, D. R., Wallace, D. R., & Gallo, A. M. (2004). Software fault interactions and implications for software testing. IEEE transactions on software engineering, 30(6), 418-421.

Name and workplace of master’s thesis supervisor:

Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D., Software Testing Intelligent Lab, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **11.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. BestounS. Ahmed Al-Beywanee, PH.D., for his help on this project and his guidance through my studies. Also, I want to thank the whole Red Hat Avocado team and especially to Cleber Rosa, who provided service from the Red Hat side.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 8. květen 2020

Abstract

Over the past decade, combinatorial interaction testing (CIT) has gain attention due to its effectiveness in finding new faults in complex systems. More specifically, CIT has been used widely to generate test suites for software systems by considering the combinations of input parameters. This form of testing known widely as $t - way$ testing, in which ($t > 2$) is the combinations strength of the inputs. In real-world software systems, constraints are present among the input parameters. Constrained CIT (CCIT) is another variant of CIT that can be used more practically nowadays. The generation of CIT and CCIT is formulated in the literature as an NP-hard problem that needs a careful design algorithm to generate new optimal solutions. Apart from the test generation, for practical applications, the tester must adapt the application and the test generation into an execution environment to run and verify the test cases. This process is complicated due to the context and nature of the different applications. This thesis presents an effort to integrate a newly developed CCIT test generation algorithm within an execution environment to help towards the flexibility of this test process. The algorithms were implemented on an industrial scale as a plugin into the Avocado test automation framework. Avocado is an open-source project maintained by Red Hat. Although the tool is popular, it doesn't have a proper CIT and CCIT test generation algorithms. CIT plugin is based on randomized meta-heuristics and uses a constrained solver to resolve the constraints among the input parameters when available. The thesis describes the technical details of the CIT plugin and its connection with the Avocado framework. It also describes its performance and limits. The new implementation of the CIT plugin shows impressive results within the Avocado framework.

Abstrakt

V posledních deseti letech si kombinatorické testování (CIT) získává pozornost díky efektivnímu nalézání chyb v komplexních systémech. CIT je široce používané ke generování testovacích scénářů pro softwérové systémy ze vstupních parametrů. Tento způsob testování je známý jako $t - way$ testování, kde $t > 2$ je síla vstupních kombinací. V reálném světě existují systémy, které mají mezi vstupními parametry podmínky. Kombinatorické testování s podmínkami (CCIT) je jedna z variant CIT, která se v dnešní době používá. Vytváření CIT a CCIT je v literatuře popsáno jako NP-těžký problém, který potřebuje speciálně navržený algoritmus pro generování optimálního řešení. Aby mohl tester testovat aplikaci, musí pro ni připravit testovací prostředí a integrovat do něj generovaní testovacích scénářů. Tento proces je velice složitý kvůli kontextu a povaze aplikace a testovacího prostředí. Tato práce přináší propojení nově vytvořeného CCIT algoritmu pro generování testů s testovacím prostředím (CIT plugin) pro zvýšení flexibility testovacího procesu. Tento algoritmus byl implementován do testovacího frameworku jménem Avocado jako plugin. Avocado je open-source projekt od společnosti Red-Hat. Ikdyž je Avocado velice populární nemá žádný CCIT ani CIT algoritmus pro generování testů. CIT plugin je založen na náhodných meta-heuristikách a používá constraint solver pro řešení podmínek mezi parametry. Tato práce popisuje technické detaily CIT pluginu a jeho propojení s Avocado frameworkem. Také popisuje jeho výkonnost a jeho limity. Nová implementace CIT pluginu přináší impozantní výsledky při propojení s Avocado frameworkem.

Školitel: Dr. BestounS. Ahmed Al-Beywanee, PH.D.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Overview

Nowadays, software applications became part of our daily lives. Almost everything that we are doing each day is connected with computers and software in some way. Within this situation, mistakes in some software applications may lead to fatal consequences and even may result in people's death. Our responsibility as software engineers is to avoid these fatal mistakes, eliminate these deadly bugs, and ensure software safety, security, and reliability. However, this may be tricky with the grown of software systems over time. Software systems grow over time dramatically. It is normal to have a system with millions of lines of code (LOC). With the expansion of the internet, the applications started to be more connected to each other, and more modular. We can imagine modern application as a system that consists of a set of interconnected applications that communicate not only to each other but also to other systems. All these things mean that software testing is more important and also is much more complicated than it was thirty years ago.

Software companies find that their success in the global market is closely connected to the quality of their products. With this ascertain, they started to be more interested in software testing. Software testing and quality assurance located not only before the software release but spans over the whole development process [1]. Software quality assurance extended into each stage of production, from the analysis of the requirements until the product's delivery to the customer. This is very important since each development phase needs a different approach and can generate diverse bugs that have to be detected as soon as possible.

With the growth of software applications, it is impossible to test everything due to the lack of time or money. Therefore the testing is not trivial. You can't create a list of every possible usage of the system and test all of these (the number of cases can be close to infinity for some software applications) [2]. Hence, a tester should mitigate the faults, risk, and failure of the system and anticipate system errors to the users [3]. The impact on users can be small, in which the users may lose their data, or it can lead to more significant problems, in which the users may lose their lives. Hence, the primary responsibility of tester is to design test cases that can reduce the possibility of fatal errors.

He has to decide which tests will be chosen from a broad set of test cases. These decisions should be smart to ensure the effectiveness of such testing based on resources and time allocated for the experiments. Efficient test cases should increase the probability of detecting system errors. To achieve such effectiveness, test design techniques help the testers to create smart test cases [4]. Each technique may used for different instances of the testing process or for detecting various types of bugs. Proper testing is necessary to combine these techniques to ensure the quality of the tested system.

1.2 Problem Statements

Software development innovations lead to the creation of larger and more interconnected systems. These innovations drive new problems in software testing. In such complex systems, unwanted interactions between system components may happened that lead to faults. The classic test design techniques were created for detecting faults caused by wrong or unwanted inputs. But they cannot securely detect faults caused by interactions between system components. Here, $t - way$ strategies can resolve this problem by generating a set of test cases that can cover the whole space of required interactions between all possible system elements.

Searching for the optimal set of t-way test cases is difficult and and considered to be NP-hard [5]. Moreover, the size of the input parameters leads to an exponential increase in computation time and complexity. Strategies based on meta-heuristic algorithms [?], show impressive results for this problem in the literature. However, they seems to have computations constraints that leads to deal with lower number of interaction strength (t). Whereas, in practice,detecting faults needs to increase the interaction strength up to t=6 in some cases. Other strategies like greedy based on Automatic Efficient Test Case Generator (AETG) [6] or the In Parameter Order (IPO[G]) algorithm [7] also showed optimized results for the test case generation. Greedy algorithms differ from meta-heuristics in how they are looking for the solution in the solution space. Unlike the meta-heuristic algorithms, Greedy algorithms don't change their choices. These features of greedy algorithms cause that they are faster than meta-heuristic, but they generate worse solutions.

In addition to the generation problem, in practice, in most of the software applications, there are constraints among the input parameters. Without considering those constraints, the generated test cases will redundant. Hence, recently, constrained interaction testing (CCIT) strategies [8] used to deal with this situation [9]. However, CCIT adds more complexity to already very complex issues of CIT. In addition to the generation complexity, the execution of the test cases is challenging due to the different running environment that must be prepared for the test execution and verification.

This thesis deals with the test generation and execution of CCIT by implementing a new plugging to an already established test automation framework developed by Red Hat called Avocado. The new CIT plugin will add new functionalities and features to the Avocado by enable the tool to take benefit of the CCIT testing method to tackle new interaction faults in the software systems.

1.3 Objectives

The aim of this project is to design and developer a new a new plugin for the Avocado testing framework, which will generate CIT and CCIT. The new plugin is designed to accept large amount of input parameters and able to find a solution for parameter size up to 200 with interaction strength up to 6. To achieve this aim, the following objective has been fulfilled:

- Investigate possible solutions to constrained interaction testing problems for a larger amount of parameters and interaction strength.
- Implement an algorithm which meet the conditions of the main aim in python.
- Connect the algorithm with the Avocado testing framework so that Avocado can generate and run the test cases.
- Test the correctness of the solution, evaluate the plugin's effectiveness and define the limitations of the developed solution.

1.4 Thesis Organization

The thesis is divided into five chapters as follows:

- Literature Background: This chapter goes through the literature, which introduces the area of software testing and combinatorial interaction testing (CIT). The chapter also gives an introduction to the fundamental test design techniques with an explanation of how important testing is in software development. Furthermore, there is a description of how CIT works and its usage with an example of combinatorial algorithms and the introduction of known constrained CIT tools. The chapter also summarise the literature of existing algorithms and strategies for solving the constrained problem in CIT. Finally, the chapter gives a description of the Avocado testing framework and how it helps the testers for test automation.

- Design and Implementation: This chapter describes how the CIT plugin designed and developed. First, the chapter explains how the Avocado plugin system works and then how CIT plugin integrated and communicates with Avocado. Next, there is an explanation of how the CIT plugin generates test suits for the Avocado and how it deals with constraints among the input configurations.

- Evaluation and Benchmarking: This chapter is about the CIT plugin's benchmarking. It shows the plugin behavior with different configurations. There is an explanation of how the CIT plugin evaluated and the evaluation results. After that, there is benchmarking of the plugin to be executed with a different condition and configurations. The evaluation results shows the effectiveness and run time of the CIT plugin and the comparison with different combinatorial testing tools.

- The fifth and last chapter is Conclusion: This chapter shows the shows the concluding remakes from the evaluation process. The chapter also shows the plugin's limitations and the future tasks which can be done in this area.

Chapter 2

Literature Background

2.1 Introduction

This chapter describes the theoretical background and concepts of CIT and the Avocado testing framework. It goes throw the literature and background related to interaction testing and brings the essential parts for understanding the topic. The chapter starts with a description of the most used test design techniques. After that, the chapter introduces CIT and how it is connected to other test design techniques. Part of this chapter is also devoted to the related definitions and notation. Finally, the chapter gives an introduction to the Avocado testing framework itself as a handy tool for system testing.

2.2 Test Case Design Techniques

One of the most important concepts in software testing is the generation of proper test cases. That is, if the tester uses poor test cases, the quality of the whole testing process will degrade. The right design of test cases is the headstone for the quality of the software testing process. Test case design techniques used to design proper test cases. This section will be dedicated to understand the well-known design test design techniques.

2.2.1 Equivalence Partitioning (EP)

In the ideal case, when we want to test a system adequately, we should consider all the conditions, e.g., every possible input. However, such a testing method is highly impractical, even impossible in some cases (e.g., the number of inputs can be infinity). The equivalence partitioning test design technique solves this problem by dividing the input data into equivalent classes that receive equal treatment. It means that if a system works for one input value, it has to work for each value from the same equivalence class. Conversely, if a test-case with a value from one class detects a bug, this bug has to be discovered by every other test-case with values from the same class. These conditions define the equivalence classes in equivalence partitioning technique. Two testers can have two different sets of classes for the same tested system if the conditions are fulfilled and the bugs are properly revealed. This approach helps a tester to test a system extensively with a smaller number of test-cases.

The main goal of equivalence partitioning is to define equivalence classes such that the number of them is the smallest and the system under test works correctly or none, for each value from one class. One of the easiest ways to define equivalence classes is to use the specification of the tested system because the specification should specify how the system has to react to different sets of data. The example in Figure 2.1 represents a simple application that calculates the type of pilot license, that the user can get, based on his flight hours.

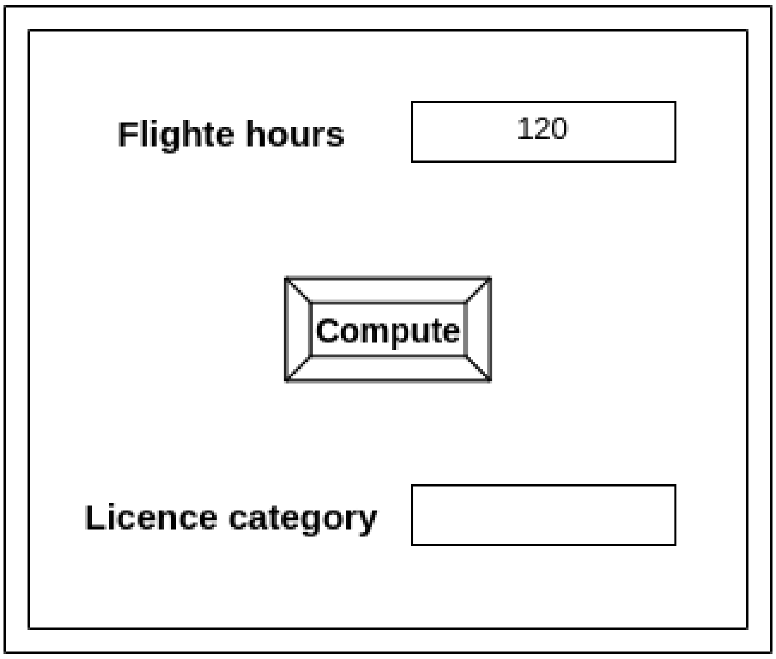


Figure 2.1: Licence computer for illustration of equivalence partitioning

For this application in Figure 2.1, if a user has less than 35 flight hours, he can't be a pilot, if he has more than 35 hours, he can be a private pilot, if he has more than 200 hours, he can be a commercial pilot. When a user has more than 1200 hours, he can be a transport pilot. Here, there is an infinite number of hours that can be used. Hence, it is impossible to test all those values. But the conditions for getting a specific type of pilot license splitting the possible flight hours into equivalence classes is possible. First the input will be 0-35, second 36-200, third 201-1200 and fourth 1201-MAX. For exhaustive testing, we can test just four values each from a different class. It will be the same as testing every possible value.

2.2.2 Boundary Value Analysis (BVA)

The boundary value analysis is connected to the EP because it works with equivalence classes, and it focuses on the boundaries between neighbor's classes. The rationale behind this test design technique is that most of the bugs occur on, above, and below the edges of equivalence classes [10]. We can look at it as equivalence partitioning creates equivalence classes and boundary value analysis picks the "right" values from each class. The truth is that the boundary value analysis creates more test cases that equivalence partitioning, but ensures better test coverage of the system under test.

Considering the example in Figure 2.1, we can see that equivalence partitioning split the data into four parts. The boundary value analysis helps to pick the values for the test cases. The boundaries are at 0, 35, 200, and 1200. In boundary value analysis we choose values (-1,0,1),(34,35,36),(199,200,201) and (1199, 1200, 1201) for the test cases.

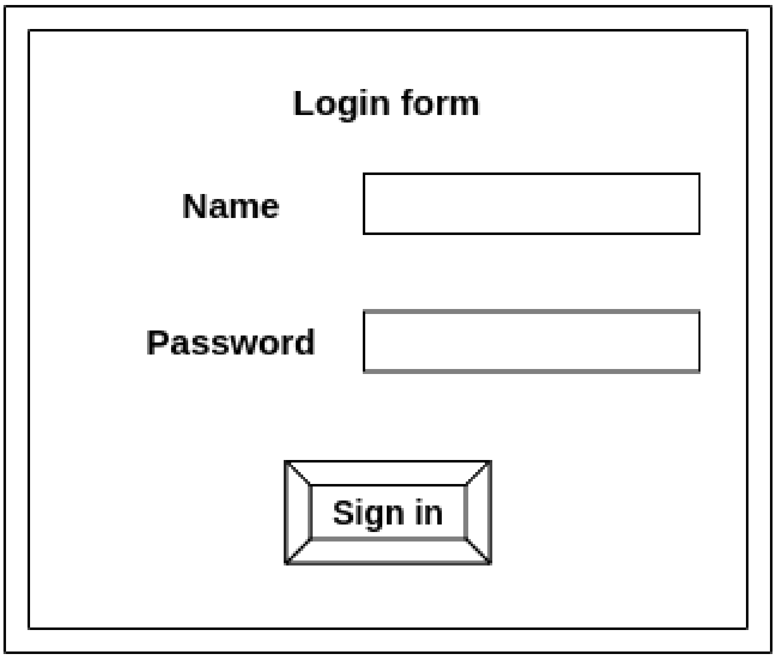


Figure 2.2: Login form for illustration of decision table

Conditions	test case 1	test case 2	test case 3	test case 4
Username	I	C	I	C
Password	I	I	C	C
Output	F	F	F	L

Table 2.1: Decision Table (C-correct, I-incorrect, F-sign in failed, L-sign in success)

2.2.3 Decision Table Testing

Unlike Equivalence partitioning and boundary value analysis, which can handle only one input at a time, the decision table can represent multiple inputs to the system and their corresponding outputs. The decision table is a testing technique that helps to test different combinations of inputs and connects these combinations with their outputs. For the determination of the correct values of each input, we can use, for example, boundary value analysis and then use the decision table for creating test-cases with multiple inputs. With the decision table technique it is easy to use and can clearly represent business logic even. The technique is helpful to both the tester and the developer to understand the business logic of the developed system that in turn prevents mistakes. The example in figure 2.2 shows a login form to a system under test. There are two inputs: Name and Password and one output: sign-in. This example does not consider the the validation of the inputs. This means that four of the inputs exist in two values: correct or incorrect. The output also has two values: success or failure of login, which depends on the correctness of the Name and Password. Whit this information, we can create a decision table represents in table 2.1. The table shows four test cases of this login form:

1. Name and Password are wrong. User can't sign-in.
2. Name is correct, but Password is wrong. User can't sign-in.
3. Password is correct, but Name is wrong. User can't sign-in.
4. Name and Password are correct. User can sign-in.

2.2.4 State Transition Diagrams

The previous methods focus on testing statically. However, the applications are dynamic, and they have many states which are changed with time. The previous techniques check each state alone and can't test transitions between system states. The technique is useful, for example, when the tester has to validate a sequence of events that can have some use case of a system under test. It can be the registration of a new user, ordering process on the e-shop, or online payment. The one input value in a specific process state can have different outputs because it depends on previous events and outputs. For such transitions between application, processes are used state transition diagrams.

These diagrams help not even in testing but also in the modeling process of system developments. The transition diagrams should be created at the start of the development process, where the application is being designed. The testers can use it then for validation of the developed application.

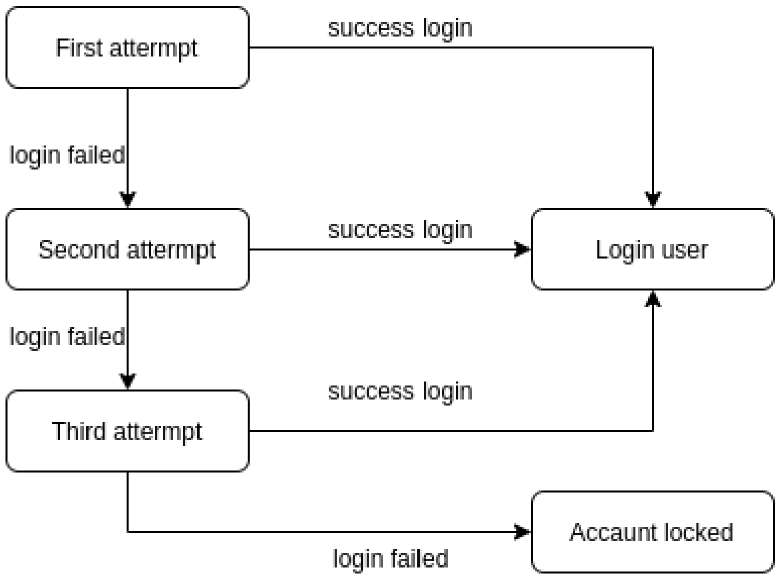


Figure 2.3: State diagram of login form

To demonstrate how the transition diagrams look like and how state transition testing works, consider the sign-in form from the previous example in figure 2.2. Consider the following use case, for security reasons we have to limit the amount sign-in attempts to three. After three unauthorized attempts the system should lock the account and inform the user about that. For such a use case, a transition diagram can be created as in figure 2.3. The diagram shows that if the user sign-in is successful, the system state is changed to log in. However,if the sign-in is unsuccessful, the state is changed to the second attempt, then to the third attempt, and finally to the locked account. This diagram helps the tester to understand how the application is designed and how it should work. We can see that in this example, there is five states and six transitions. This implies that the following test case would be the normal situation with a successful login. The second test case would be the worst situation with three wrong attempts and a blocked account. The third test case would be with one wrong attempt and then successful login. The last test case would be with two wrong attempts and then successful login.

2.3 Combinatorial Interaction Testing (CIT)

The growing complexity of modern systems leads to many new testing techniques, including the CIT [11]. CIT gain importance when the exhaustive testing is impossible and impractical due to the time and resource limitations. This problem is growing with the size and the complexity of the software under test. One approach towards test generation is to use sampling techniques to sample the system configuration and catch most of the bugs. Here, CIT is used as a sampling technique to catch the combinations among the input parameters. The following sub-sections illustrate CIT concepts clearly.

2.3.1 T-way testing

T-way testing is another variant name for CIT. It explains which interaction strength **t** is used. The interaction strength means how many parameters from configuration have to be combined in each test case. We can use CIT together with testing techniques, which are described in the previous section 2.2. For example, for a decision table with one hundred parameters, each parameter has twenty possible values. In such a case, it is almost impossible to test all combinations between these parameters because there are 20^{100} combinations that must be tested. As an alternative, CIT concepts may be used to consider 2-way or 3-way interactions [12].

As a comprehensive example, consider the testing of a web application that has to run on different machines with different web browsers and versions, different operating systems. For such a task, we have to create a table with possible values 2.2. The testing strategy must ensure that that the application works with every configuration. Testing all possible combinations of parameters requires 375 test cases in total. Here, the testing is not just run the application on the specified configuration. However, the usage of application on every configuration must be tested also. For example, imagine that we prepared around one thousand test cases for testing And we have to run these tests on all 375 configurations of the application. Here, one test case consists of one thousand tests, which means that for exhaustive testing of our application, we have to do 375 000 tests, which is impractical.

Parameters	machine	web browser	browser version	operating systems
Values	computer	Chrome	1.0.1	Windows 10
	mobile	Firefox	1.1.0	MacOS
	smartTV	Explorer	2.0.0	Ubuntu
		Edge	2.5.0	Android
		Safari	3.1.1	iOS

Table 2.2: Web application configurations

Considering the two-way interaction (pairwise) testing, only 25 test cases needed to cover the interaction, see 2.3. This approach reduces the number of test cases from 375 to 25, in our example. For instance, when we test the application, in Chrome and version 1.0.1 it works, we can rely on that and don't have to test Chrome version 1.0.1 with every machine and operating system. Higher interaction strength may used also, which raise the test coverage but also will raise the number of test cases. With the higher interaction strengh, the chance of detecting faults may increase due to the increase in the number of test cases that consider more combinations.

	machine	web browser	browser version	operating system
1	computer	Chrome	1.0.1	Windows 10
2	computer	Firefox	1.1.0	MacOS
3	computer	Explorer	2.0.0	Ubuntu
4	computer	Edge	2.5.0	Andorid
5	computer	Safari	3.1.1	iOS
6	mobile	Firefox	2.0.0	Andorid
7	mobile	Explorer	2.5.0	iOS
8	mobile	Edge	3.1.1	Windows 10
9	mobile	Safari	1.0.1	MacOS
10	mobile	Chrome	1.1.0	Ubuntu
11	smartTV	Explorer	3.1.1	MacOS
12	smartTV	Edge	1.0.1	Ubuntu
13	smartTV	Safari	1.1.0	Andorid
14	smartTV	Chrome	2.0.0	iOS
15	smartTV	Firefox	2.5.0	Windows 10
16	computer	Edge	1.1.0	iOS
17	computer	Safari	2.0.0	Windows 10
18	computer	Chrome	2.5.0	MacOS
19	computer	Firefox	3.1.1	Ubuntu
20	mobile	Explorer	1.0.1	Andorid
21	mobile	Safari	2.5.0	Ubuntu
22	mobile	Chrome	3.1.1	Andorid
23	smartTV	Firefox	1.0.1	iOS
24	smartTV	Explorer	1.1.0	Windows 10
25	smartTV	Edge	2.0.0	MacOS

Table 2.3: Example of pairwise test cases

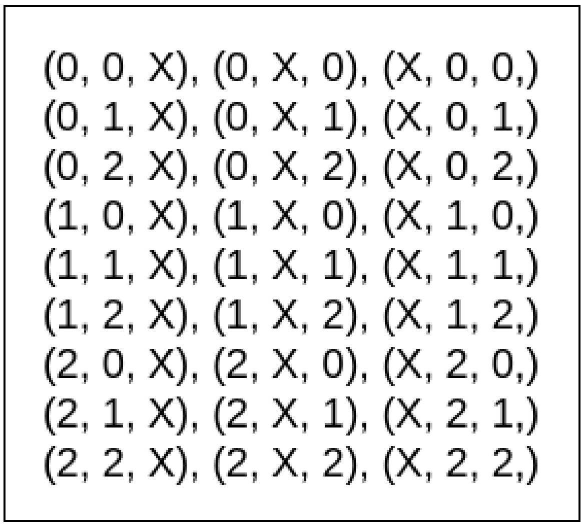

(0, 0, X), (0, X, 0), (X, 0, 0,)
(0, 1, X), (0, X, 1), (X, 0, 1,)
(0, 2, X), (0, X, 2), (X, 0, 2,)
(1, 0, X), (1, X, 0), (X, 1, 0,)
(1, 1, X), (1, X, 1), (X, 1, 1,)
(1, 2, X), (1, X, 2), (X, 1, 2,)
(2, 0, X), (2, X, 0), (X, 2, 0,)
(2, 1, X), (2, X, 1), (X, 2, 1,)
(2, 2, X), (2, X, 2), (X, 2, 2,)

Table 2.5: 2-way combinations

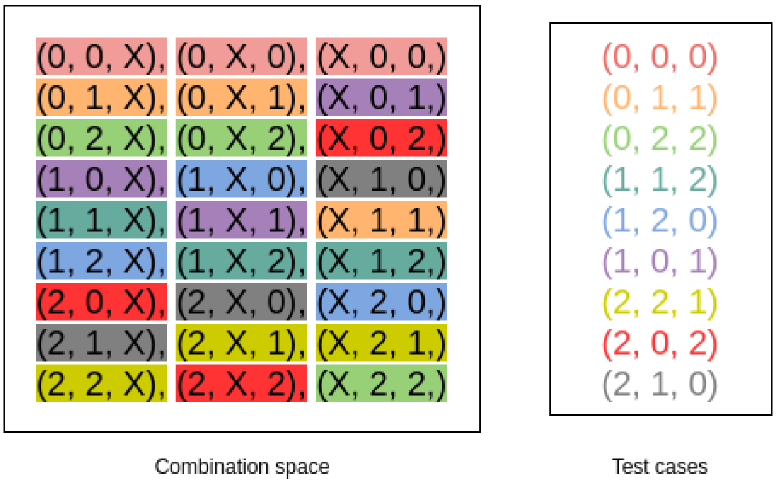
2.3.2 Interaction Coverage

The test cases for the CIT is generated with respect to the coverage of the input interactions. For a test suite with a number of test cases, the coverage should be 100% for a specific interaction strength. As a simplified example, consider a system with three parameters, each of them have three values. The complete list of parameters and values can be found in Table 2.4. Here, all possible 2-way combinations where the parameters are defined by index in combination and X means that don't care values. A set of test cases should have a complete coverage of these parameters and values.

Parameters	P1	P2	P3
Values	0	0	0
	1	1	1
	2	2	0

Table 2.4: Example of system with 3 Parameters and 3 values

The 2-way combinations mean that we have all combinations of parameters size 2. Such combinations are creating interaction space that has to be covered. One test case covers a combination if it has values equal to the a combination. Table 2.6 shows how test cases can cover more combinations and how a smaller number of combinations can cover the whole combination space. In this example, each test case covers three combinations. With bigger combinations of space and more parameters, it starts to be tricky. So the creation of the smallest test case sets is the goal of every t-way strategy.



Combination space

(0, 0, X),	(0, X, 0),	(X, 0, 0,)
(0, 1, X),	(0, X, 1),	(X, 0, 1,)
(0, 2, X),	(0, X, 2),	(X, 0, 2,)
(1, 0, X),	(1, X, 0),	(X, 1, 0,)
(1, 1, X),	(1, X, 1),	(X, 1, 1,)
(1, 2, X),	(1, X, 2),	(X, 1, 2,)
(2, 0, X),	(2, X, 0),	(X, 2, 0,)
(2, 1, X),	(2, X, 1),	(X, 2, 1,)
(2, 2, X),	(2, X, 2),	(X, 2, 2,)

Test cases

(0, 0, 0)
(0, 1, 1)
(0, 2, 2)
(1, 1, 2)
(1, 2, 0)
(1, 0, 1)
(2, 2, 1)
(2, 0, 2)
(2, 1, 0)

Table 2.6: Combination coverage

2.3.3 Meta-heuristic algorithms

Generating combinatorial interaction test suite is difficult and considered as NP-hard problem. To solve this problem, two main strategies have been used in the literature. Within those strategies either meta-heuristic algorithms or greedy algorithms have been used. To illustrate how these algorithms have been used, consider the space of all possible sets of test cases for a given test model. Such a space is called search space. The goal of a coverage strategy is to find the best feasible state. A state is feasible if it covers every combination in the combination space, as shown in section 2.3.1. The best feasible state is a state that is feasible and has the smallest size of test cases. The algorithms go through the search space exhaustively. The algorithms try to find the best feasible state or proof of its nonexistence. We can use these algorithms for small search spaces, because the exhaustive search is slow and for bigger search spaces is infeasible because of time resources. However, the bigger spaces are more often in practice [13]. For such search the meta-heuristic algorithms have been used successfully [14].

Meta-heuristic algorithms start with some initial solution that transforms into the final solution for using a randomized search for proceeding through search space [15]. The pure randomized search can find the feasible state with a probability corresponding to the number of feasible states in search space [14]. This probability can increase by some decision rules or heuristic for search. At the beginning, the search space is just a set of states that has to be defined. The two connected states are called neighbors. The search operator moves through the search space just from one neighbor to another. The heuristic of an algorithm defines how the neighbors look like and how the algorithm chooses to move from one state to another.

Simulated Annealing (SA) is an example of meta-heuristic algorithms [16]. At each step, SA randomly selects one state from the set of neighbor and evaluate its feasibility for the solution as a neighbor. The state can be reached by a change of one element. If the suitability of the selected neighbor is bigger than the current state, the neighbor becomes the new state. If the selected neighbor has smaller suitability than the current state, the heuristic of SA is used. This heuristic is modeled after the slow cooling process on the molecules of a metallic substance [17]. SA has two inputs the initial temperature $\mathbf{t_0}$ and decimal decrement factor $\boldsymbol{\delta_0}$, which is between 0 and 1.Consider the suitability of current state $\mathbf{S_0}$ and suitability of neighbour $\mathbf{S}$. Then generate a random number $\mathbf{r}$ between 0 and 1 and compute quantile $\boldsymbol{e^{(S_0-S)/t}}$. The neighbor is accepted as a new state if the $\mathbf{r}$ is less than or equal to the computed quantile. After the decision is made, the temperature $\mathbf{t}$ is multiplied by the factor $\boldsymbol{\delta_0}$. At the beginning when the $\mathbf{t}$ is close to $\mathbf{t_0}$, there is a greater probability of accepting worse neighbors, and that creates a good chance to explore more regions of the search space. After many loops of the algorithm, the probability of accepting worse neighbors decrees and the algorithm leads to the local minimum. When the algorithm finds local optimum, it stops, and this optimum becomes the solution.

2.3.4 Greedy algorithms

As stated earlier, the meta-heuristic algorithm starts with some initial solution and goes through the search space randomly, which meant that they might visit one state more than one time. On the other hand, greedy algorithms can build a solution from scratch and go through search space more systematically. They never visit one state more than one time. Due to this approach, greedy algorithms can find the solution in a shorter time than meta-heuristic algorithms. However, meta-heuristic algorithms find smaller and better solutions [18]. A typical example of a greedy algorithm for generating t-way tests is IPOG algorithm [19].

The IPOG algorithm takes the two arguments as input, the set of parameters, and $\mathbf{t}$ value. The algorithm first creates a solution for the first $\mathbf{t}$ parameters, as all combinations among those parameters. The algorithm then uses this solution to cover the t+1 parameters and iterates until the solution covers all parameters. In each iteration a new parameter is covered by tow steps horizontal growth and vertical growth. The horizontal growth adds a new column to the solution with the values of the new parameter. Each value is chosen so that it covers the largest number of uncovered t-way combinations. The vertical growth covers the remaining uncovered combinations, which can't be covered by horizontal growth. The algorithm uses two approaches, changing the existing test set or adding the new test case to the test set. The vertical growth tries to change the existing test set. To do that, it has to find values in test cases that can be changed and don't change the coverage. If the change of those values cannot cover all combinations, the vertical growth adds a new test case to the test set, which covers the largest amount of uncovered combinations.

Unlike the SA, the IPOG algorithm creates a test set more systematically. For the same set of parameters, finds the same solution every time. But because of that, it can't find some "clever" solution which can be smaller in which the SA can find it. And this is the biggest difference between greedy and meta-heuristic algorithms.

2.4 Constraint interaction testing

As illustrated previously, real systems are complex and have different nature of input parameters. Their models have to deal with constraints because not every component is connected to all other components, or the connection exists in the specific configuration. Hence, the CIT strategies must consider this situation to generate valid test cases. In general, there are two main approaches to dealing with constraints in interaction testing [20]. The first method is the usage of some constraint solver, which can validate the data. The second method is to exclude constraints from the search space. The following subsection illustrates these concepts in detail.

2.4.1 Constraint solvers

A very popular solution for constraints in CIT is the usage of constraint solvers [21]. Constraint solver is a tool that can validate data against constraints. The input is a set of constraints and some state of the system. The constraint solver answers, by true or false, if the state satisfies the entered constraints. We can use this ability within the CIT strategies. It can be used within the greedy and meta-heuristic strategies to validate the search space against a set of constraints. It can decide if the partial solution satisfies the constraints and strategy can continue searching. If the partial solution does not satisfy the constraint, the strategy has to go back and find another way through search space.

One of the most used constraint solvers with CIT strategies is SAT (Boolean satisfiability) solver [20]. The SAT problem is a significant mathematical problem that determines if a given boolean formula is satisfiable or not. The boolean formula is satisfiable only if there is an assignment of truth values to the variables that make the whole formula true [22]. A solution to this problem has numerous applications through whole computer science. One of them is constraint interaction testing (CCIT).

2.4.2 Constraints exclusion

Constraints exclusion considers the exclusion of the invalid combinations from the search space before the search process. In this method, there is no need for a constraint solver because the search space doesn't contain invalid values. Because of that, the searching is much quicker than with the usage of constraint solvers. On the other hand, it needs to have the whole search space inside memory that in turn has a significant impact on memory resources.

First, the whole search space has to be generated to create every combination of system parameters and save them into the memory. Inside this process, we can exclude the combinations which are forbidden by constraints.

With such a modified search space, greedy or meta-heuristic strategies can be used to find the solution. Both these strategies are looking for the solution inside a specified search space.

2.5 Avocado Test Framework

Avocado[1] is an open-source testing framework that runs on Python, and it is maintained by Red Hat Inc. and the Avocado community contributors. It comprises a set of tools and libraries to facilitate the creation, execution, and evaluation of automated tests. It is the successor of the Autotest framework that was designed for the Linux kernel testing [23]. It uses the best features of Autotest while mitigating Autotests' weaknesses and drawbacks. The Avocado webpage contains more detailed information about those adopted features from Autotest. The native test cases are written in Python; therefore, they follow the unit-test pattern similar to a unit test in Python. However, Avocado has a utility for running any executable as a test. So it can be used for testing every possible type of project.

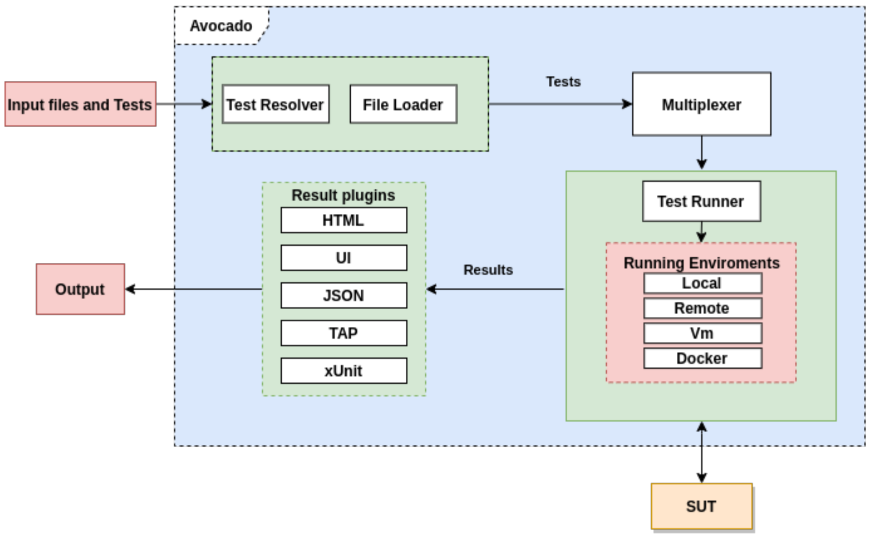


Figure 2.4: General structure of Avocado framework

As shown in Figure 2.4, Avocado comprises three main components: the test runner, libraries, and plugins. The test runner helps the users to execute test cases. Test cases can either be written in Python and use the Avocado API or in any language of the users' choice. In either case, the Avocado runner's logging system records the activities during test execution to obtain important details about the software-under-test (SUT). The primary APIs that Avocado makes available to test writers (known as "Test APIs"[2]) are derived and compatible with the Python unit test TestCase[3] class, but adds many methods for functional and performance testing. Hence, the test runner is designed to help users run and log their test cases and brings a lot of features to make their work easier.

[1]https://avocado-framework.github.io/
[2]https://avocado-framework.readthedocs.io/en/80.0/api/test/avocado.html
[3]https://docs.python.org/3/library/unittest.html#unittest.TestCase

Avocado utility libraries facilitate users to write test cases in a concise, yet expressive and powerful manner. Libraries are the most important component of Avocado; they provide several tools to speed up and simplify test development. Libraries provide extensions to Avocado by facilitating the addition of functionality and features to it. These libraries solve all types of problems, from downloading the necessary dependencies over virtual connections and networking up to debugging. They are in continual development by the community that ads new and new features.

Avocado has several plugins that can be divided into four groups: remote runner, result, variant, and testing plugins. Remote runner plugins enable users to run test cases remotely over SSH, using libvirt, or using Docker. Result plugins work with test results. They can be used for presenting and formatting the output in different formats such as JSON, xUnit, HTML, and TAP, or saving results to a database or a dedicated server. Variant plugins create variants of the test data for running different variants of the tests in the Avocado test runner. Different varianters create different test data. The CIT varianter is a variant plugin that was recently developed by us; it is an excellent outcome of industry-academia collaboration.

Testing plugins can connect different testing frameworks to the Avocado framework. When a user requires certain features that are not present in the Avocado plugins, the solution is to create a customized plugin. For that situation, the Avocado team creates a plugin system that simplifies the development process for new plugins. The user can then follow a few simple steps to create a plugin. These steps can be found in the Avocado documentation[4]. The aforementioned utilities make Avocado a powerful tool for automated testing and easily extensible to suit the varied needs of users.

[4]https://avocado-framework.readthedocs.io

Chapter 3

The Design and implementation of CIT varianter

3.1 Introduction

This chapter considers the design and implementation of the CIT plugin implemented inside the Avocado framework. Although the plugin called CIT plugin, it considers both CIT and CCIT solutions. The content of this chapter has been published in the well-known IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) [24].

3.2 CIT varianter plugin inside Avocado

The CIT varianter plugin brings advantages of CIT and CCIT (constrained CIT) to the Avocado framework [25], wherein it can apply those testing methods in a fully automated testing process. The name varianter is Avocado-specific name and inspired by the variation of the test cases. For the test generation, the user must model the system under test (SUT) parameters and constraints that will be included in the test cases. From that model, he has to create an input file in the CIT file format. Figure 3.1 provides the structure of the input file. The first part describes the parameters of the SUT and its values. The second describes constraints among the parameters' values. The constraints must be in conjunctive normal form (CNF) and use the following three operands: `!=, OR, AND`. Character || represents operand `OR` and the new-line character represents operand `AND`.

As illustrated in Figure 3.2, the CIT varianter generates from the input file, a set of test cases that comply with the input constraints and cover all $\boldsymbol{t-way}$ combinations of the parameter values for a selected level of $\boldsymbol{t}$. The $\boldsymbol{t-way}$ combination of parameter values indicates that the varianter generates a set of all combinations of size $\boldsymbol{t}$ from the parameters for each value in the set like is described in 2.3.1. The CIT varianter computes all combinations of its parameter values. Further, this set of test cases is sent to the Avocado multiplexer that creates scripts for testing the SUT.

The multiplexer can obtain several inputs from different sources for different

```
PARAMETERS
color[black,  gold, red]
shape[square, triangle, circle]
state[liquid, solid, gas]
material[leather, plastic, aluminum]
coating[anodic, cathodic]

CONSTRAINTS
color != black || shape != square
color != black || shape != triangle
color != black || shape != circle
color != gold || coating != cathodic
material != aluminum || color != gold
```

Figure 3.1: CIT input file format

usages, such as statistical data, configuration files, and testing scripts. The multiplexer recognizes the test cases originating from the CIT varianter as variants. When Avocado receives the data as variants, the test runner executes all test cases for each test case in the variants, records the output, and displays the verification message. Upon completing the testing, the runner saves the recorded results in the format chosen by the user. The format can be XML, JSON, TAP, or HTML. The runner displays the output on the screen.

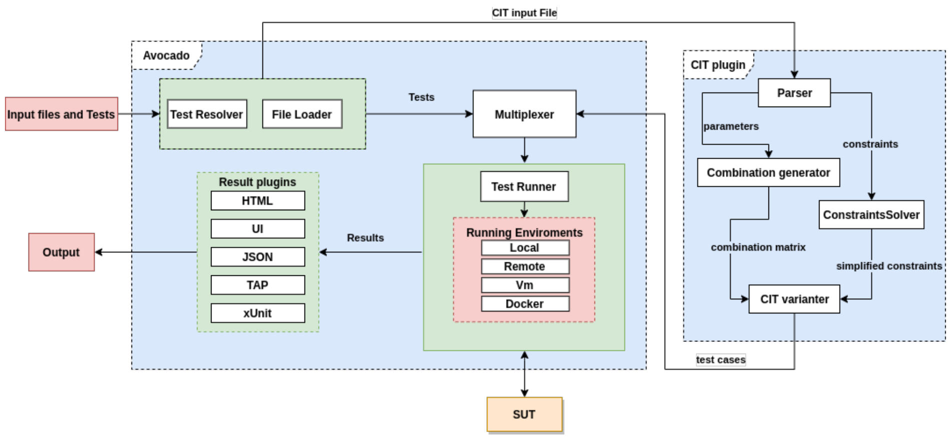


Figure 3.2: General structure of Avocado framework with CIT plugin

A significant feature of the Avocado CIT plugin is its customizable test oracle setting. Here, the user can configure the test oracle to handle the test cases automatically. Each test case is tagged by the runner as PASS or FAIL with additional information explaining the status. The status depends on the expected output of the test, and it can be customized by the user who can define the condition determining the PASS/FAIL status of the test. Here, the user can define the PASS/FAIL based on expected output. This can also be done by specifying conditions for a set of test cases rather than the individual assignment of the output per each test case. Although this adds a great feature to Avocado, research and development are needed to support automation to this feature. The advantage of using the CIT plugin with Avocado is that the test runner can use the CIT and CCIT test cases for running tests in different environments and machines that can be local, remote, or virtual. This was the description of how Avocado runs the test cases based on the CIT plugin. The following section illustrates in detail how the test generation algorithm generates test cases and deals with constraints.

3.3 Generating variants within Avocado CIT

The CIT plugin was designed, implemented, and maintained for flexible industrial usage. The aim was to use the capabilities of CIT and CCIT flexibly and practically. To this end, the CIT varianter uses a complex randomized algorithm of Monte Carlo type that finds a solution satisfying all the constraints in a reasonable time. Moreover, this solution need not be the optimal solution because it is based on the meta-heuristic strategy described in 2.3.3 and has all advantages and disadvantages of such a strategy. Algorithm 1 provides a brief description of the working of this algorithm. The algorithm can be divided into four parts, creating combinations from the search space, constraints processing, initial computing solution, and improvement of solution. The following subsections give the details of each part.

```
Algorithm 1: CIT varianter
  Input: parameters, constraints, tValue
  Output: Testcases
1 combinationMatrix = computeCombinations(data, tVvalue)
2 solver = processConstraints(constraints, tValue)
3 solver.cleanMatrix(combinationMatrix)
4 solution = computeInitialSolution(combinationMatrix)
5 while time != 0 do
6     solution = computeBetterSolution(solution, combinationMatrix)
7 Return solution
```

3.3.1 Generating the combinations

The aim of the varianter within the CIT plugin is to generate test cases that cover all $\boldsymbol{t-way}$ combinations of parameter values. To achieve this, the plugin needs to know the combinations and which test cases cover them. Consequently, the varianter creates a combination matrix, representing search space, with all $\boldsymbol{t-way}$ combinations of the parameters' values (also called t-tuples). When the varianter obtains the input file, it processes the values as numbers in a sorted matrix. Here, indexing numbers are used for the input file entries from 0 to n. For example, the parameters in Figure 3.1 are computed as `color=0, shape=1 state=2`, and the values of the color parameter would be `black=0, gold=1, red=2`.

The combination matrix uses the representation of the number of input parameters with all t-tuples. Each row of the combination matrix represents one combination of $\boldsymbol{t}$ parameters, and each column represents one combination of parameter values from a row. The value of a combination matrix cell represents how many test cases from the solution cover the exact t-tuples of parameter values. This cell can take values from -1 to the number of test cases, where -1 indicates that the combination does not exist. Figure 3.3 illustrates the combination matrix of parameters from Figure 3.1 (when t = 2) after initialization.

$$\begin{array}{c|ccccccccc}
 & (0,0) & (0,1) & (0,2) & (1,0) & (1,1) & (1,2) & (2,0) & (2,1) & (2,2) \\
\hline
(0,1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(0,2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(0,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(0,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \\
(1,2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(1,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(1,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \\
(2,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(2,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \\
(3,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1
\end{array}$$

Figure 3.3: Combination matrix initialization in Avocado CIT varianter

This is the theoretical representation of the combination matrix. For practical implementation of such a matrix is used 2D hash-table data structure, in python is represented by dictionary. The rows of the matrix are stored in a dictionary. The index of a row is represented by a tuple of parameters in which combinations are stored in the row. Another dictionary represents each row and is indexed by a tuple of parameters values. We can see that in this data structure, the matrix does not have to be full, because some rows can be bigger or smaller than others when parameters in this row have a larger or smaller amount of values. This representation was chosen because the algorithm needs to have quick access to the data, and the dictionary has time complexity for access `O(1)`. Unfortunately, where this is much quicker than other solutions, it has problems with memory consumption. Such a matrix with combinations of hundred parameters and values can take gigabytes of memory. And that can be impractical.

3.3.2 Constraints handling

The CIT varianter also handles constraints entered in the input file. A constraint can be violated, thereby leading to a set of invalid and non-executable test cases. Figure 3.1 shows an example of constraints. It can be seen that these constraints forbid the test cases with black color and square shape, black color and triangle shape, etc. However, constraints in real-world scenarios such as those on the industrial scale can be considerably complicated, and the varianter must process them efficiently. Moreover, the format to represent these constraints in the input file must be CNF [26] for the ease of constraint computation and use. However, despite the CNF format of the constraints, the varianter must preprocess them to simplify them or find some implicit constraints that are not explicitly mentioned in the input file but are defined by other constraints. For example, in Figure 3.1, the first three constraints define that the color cannot be black because every shape cannot have black color. This means that we can reduce these three constraints to one.

It is not practical to let the user specify all the constraints in the SUT. However, it is possible to generate all the implicit constraints based on a small number of explicit ones. For preprocessing the constraints, the varianter implemented the constraint handling mechanism for handling forbidden tuples presented in [27]. To transform the input constraints into forbidden tuples, we consider the CNF because each disjunction part of CNF represents a forbidden tuple. For example, we can transform the constraints in Figure 3.1 into a set of five forbidden tuples as shown in Figure 3.4. The varianter derives a set of forbidden tuples and uses it for constraints validation. A test case is valid if and only if it does not contain any forbidden tuple.

```
{{color=black, shape=square},
{color=black, shape=triangle},
{color=black, shape=circle},
{color=gold, coating=cathodic},
{material=aluminum, color=gold}}
```

Figure 3.4: Forbidden tuples form the input file

These properties work precisely with the data representation in the combination matrix. The CIT varianter does not use constraint solver for validating constraints as in 2.4.1, but precompute the search space represented by the combination matrix that it meets the constraints as in 2.4.2. In other words, every combination that contains a forbidden tuple is forbidden. Based on this mechanism, the varianter finds the forbidden cells inside the combination matrix and tags them as -1, thereby indicating the "does not exist combination." This process is called matrix cleaning. However, before the varianter cleans the combination matrix, it must compute the forbidden tuples from the input constraints. For this computation, the varianter uses two methods: **derive**, for finding new implicit tuples from the existing set and **simplify**, for the reduction of unnecessary tuples from the set.

The **derive** method uses one rule to derive the implicit forbidden tuples from existing ones. If we have parameter P with values n then there are n forbidden tuples, where each tuple contains different values of P. The varianter constructs a new forbidden tuple by combining all values in these n tuples except the values of parameter P. Using this rule, the **derive** method finds all forbidden tuples for each parameter. Furthermore, if the rule condition is satisfied, this method creates a new forbidden tuple. In Figure 3.4, the rule condition is fulfilled for parameter *shape*, and it is evident that the new forbidden tuple is *color=black*.

Algorithm 2: Constraints processing

```
Input: constraints, combinationMatrix
Output: combinationMatrix
1  size = size(constraints)
2  derive(constraints)
3  simplify(constraints)
4  while size != size(constraints) do
5      size = size(constraints)
6      derive(constraints)
7      simplify(constraints)
8  for $i = 0$ to combinationMatrix.length, $i + 1$ do
9      if combinationMatrix[i] is in constraints then
10         combinationMatrix[i] = -1
11 Return combinationMatrix
```

The **simplify** method finds the tuples that can be removed from the set while ensuring that the results of constraint validity will not be affected. Here, a forbidden tuple that contains another tuple from the set can be removed because any test satisfying a tuple must cover the subset of that tuple. In our example, the **derive** method finds tuple *color=black*, which is a subset of each of the first three tuples. This means that the first three tuples can be removed.

```
{{color=black},
{color=gold, coating=cathodic},
{material=aluminum, color=gold}}
```

Figure 3.5: Forbidden tuples after preprocessing

A single call to the **derive** and *simplify* methods cannot ensure that the transformation of constraints into forbidden tuples is complete because the **derive** method generates new implicit tuples that can be used for generating other implicit tuples. As you can see in Algorithm 2 the varianter runs these methods in a loop, and the loop ends when the methods cannot change the

$$\begin{array}{c|ccccccccc}
 & (0,0) & (0,1) & (0,2) & (1,0) & (1,1) & (1,2) & (2,0) & (2,1) & (2,2) \\
\hline
(0,1) & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
(0,2) & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
(0,3) & -1 & -1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
(0,4) & -1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & -1 \\
(1,2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(1,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(1,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \\
(2,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(2,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \\
(3,4) & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1
\end{array}$$

Figure 3.6: Cleaned combination matrix

forbidden tuples any more, thereby indicating that the tuple transformation is complete. Subsequently, the varianter generates a complete forbidden tuple set and cleans the combination matrix using the forbidden combinations. Figure 3.5 presents the set of tuples from Figure 3.4 after this process, and the cleaned combination matrix is illustrated in Figure 3.6. Such a matrix represents search space, which meets the constraints.

3.3.3 Generation of an initial solution

Post the cleanup of the combination matrix based on the generated forbidden tuples; the combination matrix comprises valid combinations that must be covered. The varianter generates an initial solution that is the set of test cases that cover the valid combinations (i.e., $\boldsymbol{t - tuples}$). At this stage, the aim is to find any solution that covers all combinations, without considering its optimality. The generation of the initial solution involves processing the test cases iteratively until the combination matrix is fully covered. The pseudocode to generate the initial solution is presented in Algorithm 3.

The random search mechanism of the Avocado plugin is, in fact, not entirely random. Here, we followed the "Hamming distance" mechanism presented in [28]. Only the first test cases are generated randomly, whereas the remaining are created from the first ones. During the random search, the varianter first creates two random test cases valid as per the constraints. This validity is ensured the same as in the combination matrix. The new random test case must not contain any forbidden tuple from constraints generated in constraint processing. Subsequently, for each test case, the varianter computes the Hamming distance from an already found solution and chooses the one with the largest Hamming distance. Then, this test case is added to the solution. The Hamming distance between the test case and the initial solution is the sum of the Hamming distances between the test case and each test case of the initial solution. The random search generates new test cases until the number of the uncovered tuples is smaller than the number of tuples covered by more than one test case. At this point, the search process becomes inefficient because the probability of covering a new tuple is smaller than the probability of covering an already covered tuple. In this scenario, the varianter must complete and fill the initial solution by a discrete method for covering the tuples. This method uses all uncovered

Algorithm 3: Solution initialization

```
Input: combinationMatrix
Output: Testcases
solution = [ ]
combinationMatrix = computeCombinations(data, tValue)
while uncoveredNumber != 0 do
    if uncoveredNumber > coveredMoreThanOnesNumber then
        $testCase_1$ = randomTestCase
        $testCase_2$ = randomTestCase
        if distance($testCase_1$,solution)>distance($testCase_2$,solution)) then
            testCase = $testCase_1$
        else
            testCase = $testCase_2$
    else
        testCase = coverMatrix()
    combinationMatrix.coverCombinations(testCase)
    solution.add(testCase)
Return solution
```

tuples from the combination matrix, chooses the largest disjunct set from them, which will be covered by the next test case. Some of the values from the new test case maybe not defined by the disjunct set. These values are filled randomly by the varianter. This discrete method adds new test cases to the initial solution until all tuples are covered and the initial solution is complete. Figure 3.7 shows a sample solution where the rows denote test cases, and the columns denote parameters of the SUT.

$$\begin{pmatrix} 1 & 2 & 1 & 1 & 0 \\ 2 & 2 & 1 & 2 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 0 & 0 \\ 1 & 1 & 2 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 2 & 2 & 2 & 0 & 1 \\ 2 & 0 & 1 & 2 & 1 \\ 2 & 0 & 0 & 2 & 0 \\ 1 & 2 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 2 & 1 & 2 & 2 & 1 \\ 1 & 0 & 2 & 1 & 0 \end{pmatrix}$$

Figure 3.7: Initial solution

3.3.4 Improvement of the solution

Eventually, the set of test cases covering all t-tuples of parameter values created by the varianter becomes available. This set can be used for testing the SUT. However, this set of test cases is not optimal and it is possible to cover the combinations with a smaller set of test cases. The CIT varianter aims to find the smallest possible valid solution at a specified time.

To this end, the next stage of the varianter algorithm is to optimize the size of the initial set of test cases while maintaining the coverage of all tuples. Here, the CIT varianter uses the mixed neighborhood tabu search (MNTS) algorithm described in [28]. This algorithm is based on the randomized tuning of the solution that leads to a better solution. As shown in Figure 3.7, MNTS represents the solution as a matrix of size $\boldsymbol{N} \times \boldsymbol{M}$, where each row represents one test case and each column represents one input parameter of the SUT. MNTS aims to delete the specified number of rows from the solution matrix. Then, using the randomized algorithms, the new solution is modified to cover the combination matrix. In other words, the tuple coverage level is maintained, covering all tuples with a smaller number of rows (i.e., test cases) in the matrix. Algorithm 4 provides the pseudocode and the procedure of this algorithm.

Algorithm 4: Solution improvement

```
   Input: solution, combinationMatrix
   Output: Testcases
1  M = 600
2  newSolution = solution.copy()
3  combinationMatrix = computeCombinations(data, tVvalue)
4  deletedRow = newSolution.deleteRow()
5  combinationMatrix.uncoverCombinations(deletedRow)
6  for $i = 0$ to M, $i + 1$ do
7      algorithm = choose(1,2,3)
8      newSolution = algorithm(newSolution)
9      combinationMatrix.updateCoverage(newSolution)
10     if uncovered == 0 then
11         Return newSolution
12 Return solution
```

The algorithm initially deletes a random row from the solution matrix and uncovers the t-tuples of the deleted test cases. Uncovering the t-tuples means decreasing by 1, every cell in the combination matrix covered by the deleted test case. The solution that is not complete at this point must be modified to cover the entire combination matrix. For every modification to the solution matrix, one of the three modification functions N_1, N_2, and N_3 is randomly chosen with probability 0.1, 0.1, and 0.8, respectively. This probability ratio is based on the experiments described in [28]. Each modification function modifies the solution matrix to create a new matrix, which is achieved by changing the numbers in the randomly selected cells. The modification with the best coverage of the combination matrix is selected as a new solution matrix for other modifications. N_1 randomly selects a cell of the solution matrix and makes all possible changes to the cell number. N_2 randomly selects a column in the solution matrix and changes each cell in this column. N_3 randomly selects one of the uncovered combinations to form the combination matrix and changes every row in the solution matrix to cover this combination. When the modified solution covers the entire combination matrix, the algorithm executes the initial step of deleting another row from the matrix and continues. The algorithm defines the constant, m, that represents the maximum number of modifications to the solution. We have chosen the maximum number of modifications based on our experimental experience and preferences and section 2.3.2 is talking about that. However, the user has the option to change this number based on the application preferences. When the number of modifications reaches m, it is concluded that a better solution cannot be found, and the last complete solution is treated as the best solution. This number influence this size of solution and the run-time of varianter. If this number is small, the varianter finish earlier, but with a larger solution.

Chapter 4

Evaluation

4.1 Introduction

This chapter introduces the description and results of the experiments on the performance of varianter. The experiments was testing different technical areas. We created four experiment methods. Each one focused on different area of the CCIT. As a first step, we defined the default number of iterations inside the CIT algorithm 3.3.4. The second experiment focused on testing of varianter limitations. In order to find out the varianters memory problem, this experiment is focused on measuring memory consumption. The third experiment, measured the runtime and test-case size of the varianter. The last experiment is targeted to the constraint problem and measure how constraints affect the varianter performance. These measurements are focused on the performance of CIT varianter as part of the Avocado test framework. The experiments do not measure the efficiency of faults detection of Avocado on some SUT.

4.2 Experimental device description

To run all the experiments, the notebook Thinkpad T480s was used with the following configuration:

- OS - Fedora 32 (Workstation Edition)
- OS type - 64-bit
- GNOME version 3.36.3
- Memory - 15 GB
- Processor - Intel® Core™ i7-8650U CPU @ 1.90GHz × 8
- Graphic - Mesa Intel® UHD Graphics 620 (KBL GT2)

4.3 Iterations benchmarking

It is necessity to define the iteration constant m which is described in section 3.3.4. The constant m influences size of the solution and runtime of the varianter and it is necessary to be defined before the performance evaluation. The user can set this constant for his special use-cases, but it has to be defined a default value, which ensures the right function of the varianter in most cases. To ensure that, we created the following experiment. The experiment model has 30 parameters; each parameter is in range of two to five values. This model was constrained from 20%, where one constraint influences maximal three parameters. We started with m = 0 and increased it up to 1000 by 100 steps. Hence the CIT algorithm is meta-heuristic and randomized, the result above the same model can differ. Knowing this, we run each configuration five times, and as a result, we introduce the average value of these five tests.

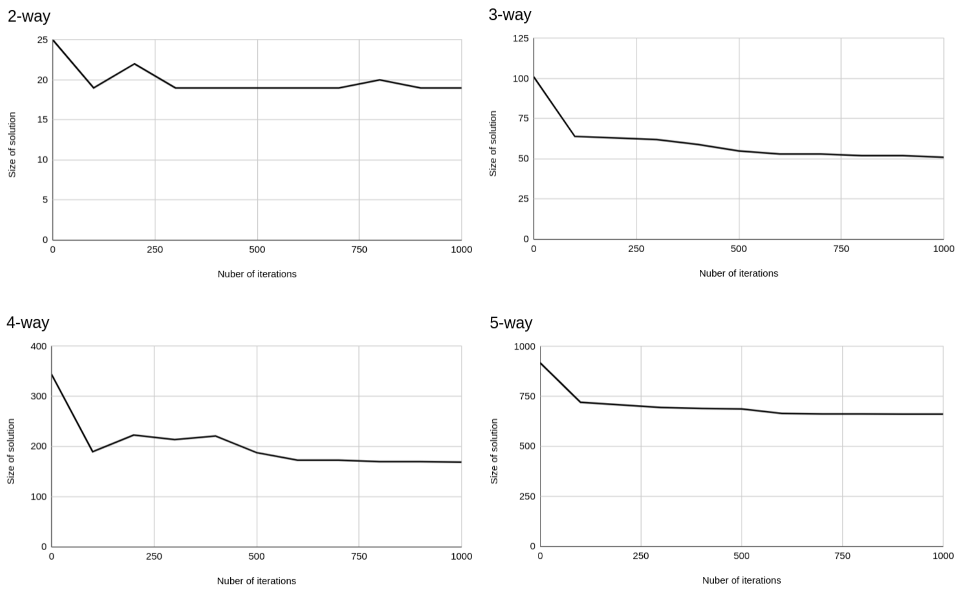


Figure 4.1: Iterations benchmarking for 2-way, 3-way, 4-way, 5-way testing

Figure 4.1 shows the results for 2-way, 3-way, 4-way and 5-way of the evaluated model. Based on this data, we can see the size of initial solution form section 3.3.3 (here represented by the number of iterations = 0) and how the CIT algorithm can improve the solution size 3.3.4. It is also clear that the solution evolves due to the m. According to those figures, we set the default value m to 600. The reason is that crossing this value, the size of the solution is almost constant.

4.4 Memory consumption

The second experiment is focused on CIT varianters limitations. We have to find out the maximum number of parameters/values that the varianter can work with. Also, which models are out of range. The varianter computes constraints by constraints exclusion 2.4.2. It is necessary to have the complete search space stored inside the memory (represented by combination matrix). This has significant impact on the memory consumption. Size of combinations can be around gigabytes of data, and that's why the memory is the bottleneck of the CIT varianter.

Based on those findings, we measured the memory consumption of CIT varianter. We created test models with two values parameters. Number of parameters in each model is decreased by 10. The experiment computes models form 10 to 300 parameters with interaction strength t up to 6. These tests were run just once because the memory is not affected by randomization of the algorithm.

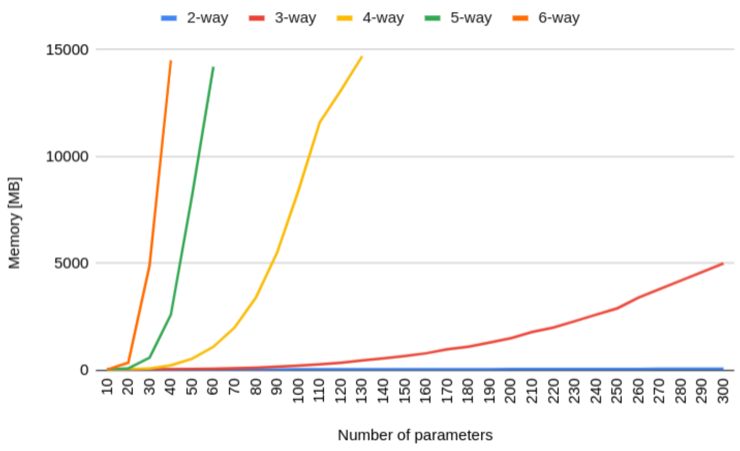


Figure 4.2: Memory consumption of CIT varianter fot 2-way, 3-way, 4-way, 5-way and 6-way testing

Figure 4.2 shows the memory consumption of CIT varianter for integration strength up to 6. This graph visualize that the size of the memory grows with the growth of the combination matrix size. On the used device, there was 14,5 Gb memory free for computations. This experiment discovers that the CIT varainter was able to compute test cases for all models for 2-way and 3-way testing. For 4-way testing, it was able to calculate models of up to 140 parameters, which means that the biggest combination matrix was 181 742 080 combinations. For 5-way testing, it was able to calculate models of up to 60 parameters, which means that the biggest combination matrix was 174 768 384 combinations. For 6-way testing, it was able to calculate models for up to 40 parameters, which means that the biggest combination matrix was 245 656 320 combinations. Based on this results, we can see that for the used device with useful memory around 14,5 GB, the limitations of

CIT varianter are around 200 000 000 combinations.

4.5 Varainter bench-marking

In the third experiment, we measure the actual efficiency of the CIT varainter. This experiment's target is to measure the number of test cases generated by the varianter for different models. The purpose of this experiment is also to measure the run-time of computing these test cases. The test is very similar to memory consumption experiment 4.4. We use the same models with two values in a parameter. We start with 10 parameters in the first evaluation round and end with 300 parameters. The number of parameters is decreasing by 10. It is important to mention that, we can not compute all these tests for each t-way testing, because of the memory consumption 4.4. Knowing this we measure only tests that can be computed with CIT varianter on our machine. The results of this experiment is visible in tables 4.2 and 4.1.

stregth test	4		5		6	
-	run-time [s]	test-cases	run-time [s]	test-cases	run-time [s]	test-cases
1	58	24	486	68	511	146
2	4759	56	5921	139	10586	723
3	7266	72	9986	577	33578	2567
4	8759	83	21769	1346	98730	4926
5	9841	156	56812	2507	X	X
6	12498	179	95351	4578	X	X
7	16581	227	X	X	X	X
8	21598	272	X	X	X	X
9	31746	328	X	X	X	X
10	54416	372	X	X	X	X
11	83576	403	X	X	X	X
12	92351	441	X	X	X	X

Table 4.1: Varainter bench-marking 4-way, 5-way and 6-way

test/stregth	2		3	
-	run-time [s]	test-cases	run-time [s]	test-cases
1	1,16	6	1,15	12
2	3,43	8	15,98	21
3	4,61	8	79	26
4	9,52	9	146	29
5	18,28	9	323	31
6	25,32	10	904	34
7	31	10	1555	36
8	41	10	2373	38
9	47	10	3524	39
10	126	10	4484	40
11	167	10	6152	41
12	120	10	8162	42
13	207	11	8690	44
14	183	11	9659	44
15	268	11	10038	46
16	365	11	10370	47
17	358	11	11967	50
18	372	11	12369	50
19	404	11	12757	51
20	540	11	13200	53
21	641	12	14357	55
22	860	12	15010	55
23	726	12	15996	57
24	1235	12	16772	58
25	1011	12	18908	59
26	929	12	22251	61
27	1635	12	24527	61
28	1353	12	27059	63
29	1639	12	29126	66
30	2169	12	32678	68

Table 4.2: Varainter bench-marking 2-way and 3-way

4.6 Constraints influence

In the last experiment, we measure how the constraints influence the behavior of the CIT varianter. Dealing with constraints is a significant feature of the varianter. It is necessary to know how computing is changed due to the size of constraints in the model. We prepared ten models, each of them has ten parameters with two values. The models are different just in the size of constraints. First test is not constrained at all. The second has constrained 10% of values. The third is constrained from 20%. And this is going up to 90% in the last test. In this experiment, we measure the runtime of the CIT

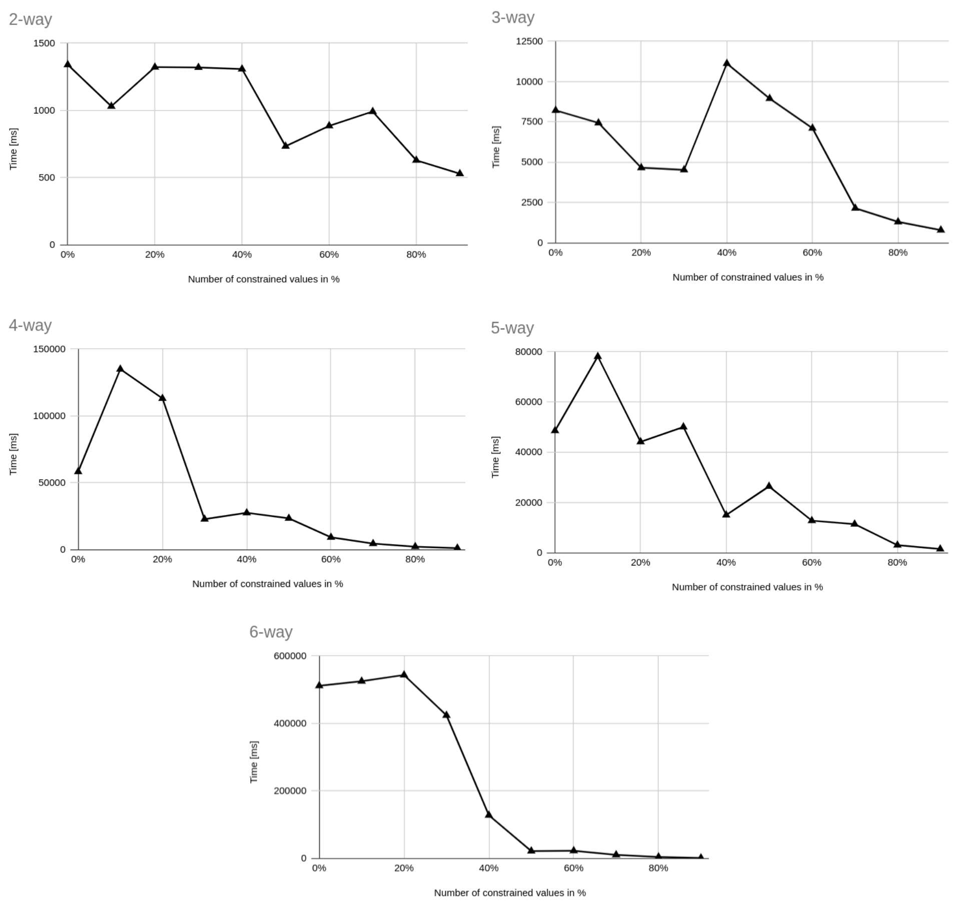


Figure 4.3: Constraints influence for 2-way, 3-way, 4-way, 5-way and 6-way testing

varianter on these tests. Because the varainters algorithm is randomized, each test runs five times. As a result we take average runtime of these tests.

Figure 4.3 shows the decreasing trend of runtime against the number of constraints inside the model. The reason is the decreasing size of possible combinations inside the model. Each constraint removes some combinations from the combination matrix. That makes the matrix smaller and searching faster 3.3.2. At the beginning of the experiment (where the models are constrained by 10 or 20 percent) is visible the overload of constraint solving to the CIT plugin. This data shows us that when we have systems with a smaller number of constraints, we have to anticipate with overload in computing. The overload becomes to be unimportant when the constraints grow.

Chapter 5
Conclusion

5.1 Introduction

This thesis introduces the CIT tool for the Avocado testing framework called CIT plugin. The objective was to create a tool for Avocado that can handle models with many parameters and can computes with combination strength up to 6. This chapter summarises the earlier chapters and discusses the achieved results and describes the plans for this tool.

5.2 Overview

Chapter 1 introduces the problem of software testing. It explains the importance of software testing in the developing process. Part of this chapter was focused on history of software testing and it's evolution. At the end there an introduction to the problem that this thesis aims to solve: computing large systems with higher connectivity.

Chapter 2 is about a literature background. It describes the most basic test case design techniques such as equivalence partitioning and boundary value analysis. These techniques describe the way in which software testing went to CIT. It explains what interaction testing is and how it works. It describes different approaches in the interaction testing such as greedy and meta-heuristic strategies. There is a comparison of advantages and disadvantages of these strategies and how they support solution of our problem.

Chapter 3 brings the details of the CIT plugin and Avocado framework. It describes how the Avocado framework works and how the CIT plugin is connected to it. Illustrates each part of CIT computation and how it is connected to solve the CIT problem.

Chapter 4 defines experiments on CIT plugin. It consists of several experiments that test the CIT plugin in different areas, such as memory consumption, run-time, and constraints influence. Base on the results of these experiments, we can discuss the effectiveness and use-ability of the CIT.

5.3 Discussion

After showing the implementation and the evaluation results, it can be concluded that the new plugin for Avocado can generated effective test cases for models with many parameters. For larger combination strength, we are facing memory consumption problems. Base on this the CIT plugin can compute models in size of tens parameters.

The experimental evaluation showed impressive results. It is visible that CIT plugin can decrease the number of test cases needed for covering all possible combinations of the system. Those results shows a little high run-time of computing. This is not a problem when we have complex systems with hundreds and thousands of tests that will run repeatedly. Considering the results, we can make the conclusion that the goal was achieved.

5.4 Future plans

Considering the results from chapter 4, we can set plans for the next development. The result shows us the way how to compute big models with bigger interaction strength by solving the memory issue problem. This could be done by splitting the combination matrix into the different parts and compute them separately. This is challenging problem with lot of road blocks. This will be the main task to solve for the next development. The next possible improvement will be the parallelization of computing. The process of searching through the combination matrix in section 3.3.4 it can be paralleled. It can decrease the run-time of the computation. When we deal with the problem of splitting the combination matrix into different parts, we can parallels this process as well. These are the main tasks for improving the CIT plugin, and it will need more research to do that.

Bibliography

[1] L. Baresi and M. Pezze, “An introduction to software testing,” *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 89–111, 2006.

[2] N. Ashrafi, “The impact of software process improvement on quality: in theory and practice,” *Information & Management*, vol. 40, no. 7, pp. 677–690, 2003.

[3] A. Mette and J. Hass, “Testing processes,” in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE, 2008, pp. 321–327.

[4] J. A. Whittaker, *Exploratory software testing: tips, tricks, tours, and techniques to guide test design.* Pearson Education, 2009.

[5] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, “Combinatorial software testing,” *Computer*, vol. 42, no. 8, pp. 94–96, 2009.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, “The aetg system: An approach to testing based on combinatorial design,” *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[7] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, “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).* IEEE, 2007, pp. 549–556.

[8] I. H. Hasan, B. S. Ahmed, M. Y. Potrus, and K. Z. Zamli, “Generation and application of constrained interaction test suites using base forbidden tuples with a mixed neighborhood tabu search,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 30, no. 03, pp. 363–398, 2020.

[9] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, “Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading,” *Information and software Technology*, vol. 86, pp. 20–36, 2017.

[10] D. Abdullah and I. Burnstein, “Practical software testing,” 2003.

[11] B. S. Ahmed, A. Gargantini, K. Z. Zamli, C. Yilmaz, M. Bures, and M. Szeles, “Code-aware combinatorial interaction testing,” *IET Software*, vol. 13, no. 6, pp. 600–609, 2019.

[12] J. Czerwonka, “Pairwise testing in real world,” in *24th Pacific Northwest Software Quality Conference*, vol. 200. Citeseer, 2006.

[13] D. L. Kreher and D. R. Stinson, “Combinatorial algorithms: generation, enumeration, and search,” *ACM SIGACT News*, vol. 30, no. 1, pp. 33–35, 1999.

[14] J. Stardom, *Metaheuristics and the search for covering and packing arrays.* Simon Fraser University Burnaby, 2001.

[15] A. B. Nasser, K. Z. Zamli, A. A. Alsewari, and B. S. Ahmed, “An elitist-flower pollination-based strategy for constructing sequence and sequence-less t-way test suite,” *International Journal of Bio-Inspired Computation*, vol. 12, no. 2, pp. 115–127, 2018.

[16] P. J. Van Laarhoven and E. H. Aarts, “Simulated annealing,” in *Simulated annealing: Theory and applications.* Springer, 1987, pp. 7–15.

[17] E. Falkenauer, *Genetic algorithms and grouping problems.* John Wiley & Sons, Inc., 1998.

[18] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, “Constructing test suites for interaction testing,” in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 38–48.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, “Ipog/ipog-d: efficient test generation for multi-way combinatorial testing,” *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[20] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, “Constrained interaction testing: A systematic literature study,” *IEEE Access*, vol. 5, pp. 25 706–25 730, 2017.

[21] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, “Constrained interaction testing: A systematic literature study,” *IEEE Access*, vol. 5, pp. 25 706–25 730, 2017.

[22] T. Walsh, “Sat v csp,” in *International Conference on Principles and Practice of Constraint Programming.* Springer, 2000, pp. 441–456.

[23] J. Admanski, “Autotest – testing the untestable,” in *Proceedings of the Linux Symposium*, 2010, pp. 9–18.

[24] J. Richter, B. S. Ahmed, M. Bures, and C. R. R. Junior, “Avocado: Open-source flexible constrained interaction testing for practical application,” *arXiv preprint arXiv:2002.00390*, 2020.

[25] B. S. Ahmed, A. Pahim, C. R. R. Junior, D. R. Kuhn, and M. Bures, “Towards an automated unified framework to run applications for combinatorial interaction testing,” in *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 252–258.

[26] J. P. Warners, “A linear-time transformation of linear inequalities into conjunctive normal form,” *Information Processing Letters*, vol. 68, no. 2, pp. 63 – 69, 1998.

[27] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, “Constraint handling in combinatorial test generation using forbidden tuples,” in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2015, pp. 1–9.

[28] L. Gonzalez-Hernandez, “New bounds for mixed covering arrays in t-way testing with uniform strength,” *Information and Software Technology*, vol. 59, pp. 17 – 32, 2015.

List of Attachments

1. Source code of Avocado framework with CIT plugin
2. CIT plugin installation guide.