An Analysis and Extension of Category Partition Testing in the presence of Constraints

By

Sunint Kaur Khalsa

A thesis submitted to
The Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
(OCIECE)
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada K1S 5B6

March 2017

The undersigned recommend to
The Faculty of Graduate and Postdoctoral Affairs
the acceptance of the thesis

An Analysis and Extension of Category Partition Testing in the presence of Constraints

Submitted By
Sunint Kaur Khalsa

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dr. Iain Wallace, Chair Examination Committee, Carleton University

---

Dr. Yvan Labiche, Thesis Supervisor and Department Chair, Carleton University

---

Dr. Hélène Waeselynck, External Examiner, LAAS-CNRS, France

Dr. Jean-Pierre Corriveau, Internal Examiner, Carleton University, Ottawa

---

Dr. Lucia Moura, University of Ottawa, Ottawa

Dr. Chung-Horng Lung, Carleton University, Ottawa

Carleton University, 2017

Abstract

Category Partition (CP) is a Black Box testing technique that formalizes the specification of the input domain of the system under test. A CP specification is driven by the tester's expertise and comprises of parameters, categories (characteristics of parameters) and choices (acceptable values for categories) required for extensively testing the system. To ensure completeness, choices not only correspond to permitted input values but also correspond to values that account for boundaries or robustness. These choices are combined on the basis of various selection criterion (e.g., Base Choice, Pairwise) to form test frames which given input values form test cases.

To ensure that the combinations of choices are feasible and account for valid sets of user requirements, constraints are introduced. A constraint can be a specification for permitted choice combination or a choice annotation as Error or Single. In a typical development environment where testing is driven by stringent deadlines, a tester might have to decide how many constraints (constraints among choices, Error or Single annotation) are enough to attain the maximum level of test completeness. The present work will assist a test engineer in making this decision. This thesis contributes by concluding, based on experimental evaluation of academic and industrial case studies, that in case of limited resources an equally effective test suite can be attained by meticulously defining Error and Single annotations in a CP specification. The present work also contributes by challenging the notion that introducing constraints reduces the cost of the test suite by restricting the combination of choices. This thesis asserts, based on experimental evaluation, that introducing constraints does not always reduce cost and that the cost of the test suite depends on various other factors.

Combining choices while accounting for constraints and forming each choice adequate test set, for instance, is achievable, thanks to Constrained Covering Arrays from combinatorial testing. But, Base Choice (BC) criterion besides being an integral and domain specific criterion has not been defined to specifically account for constraints on choices. This thesis contributes by introducing two extensions to the BC criterion: Constrained Base Choice and Extended Constrained Base Choice, to account for constraints resulting from complex input domain. A number of academic and industrial case studies have been used to compare different selection criteria, including the new ones, in terms of data flow coverage, control flow coverage, cost and effectiveness at finding faults. Results show the performance of the new criteria equivalent to a 3-way selection criterion with a much smaller cost.

The use of CP on large systems can result in a large number of parameters (categories) and values (choices). A technique based on combinatorics called Combinatorial Testing (CT) is then used to automate the process of creating those combinations of choices to form test frames. CT is typically performed with the help of combinatorial objects called Covering Arrays. Another contribution of this thesis is the survey and compilation of thoroughly searched 75 algorithms and tools for generating a combinatorial test suite. These algorithms and tools are categorized based on the features specific to the CP technique, which includes tool's support for selection criteria, mixed covering array, constraints etc. Results can be of interest to researchers or software companies who are looking for a CT algorithm/tool suitable for their needs.

Dedication

Every *Shabad* (word) in this thesis is dedicated to my *Shabad Guru:*

Sri Guru Granth Sahib Ji;

the holy scripture of the Sikhs, the eternal living Guru.

Acknowledgments

"If I have seen further it is by standing on the shoulders of Giants"

- Sir Isaac Newton

I am honored and feel blessed while talking about the 'Giants' whom I am fortunate to rely and lean upon.

First and foremost, My *Akalpurakh*, the Almighty God. I have lost count of the immense blessings I have been showered upon, with each passing minute. Thank you God for this life and for the strength and capability to flourish.

I am indebted to my thesis supervisor, Dr. Yvan Labiche. I am short of words to express my gratitude for your incredible help, support, encouragement and financial support throughout my PhD program. Your constructive criticism and your amazing mentorship have been the stepping stones to get me this far. I owe to your patience, to your attention to fine details, to your long days, to your brainstorming and brain stirring discussions which not only encouraged me to push my boundaries and think out of the box but also taught me the intricacies of good technical writing. Besides being a supervisor for my academic research, you have also taught me the essence of empathy; your words "people are more important than thesis" keeps resonating in my mind and helps me make decisions in times of dilemma. Thank you Prof Labiche for making this journey a truly memorable and integral part of my life.

I would like to thank my thesis committee members Dr. Hélène Waeselynck, Dr Jean-Pierre Corriveau, Dr. Lucia Moura, Dr. Chung-Horng Lung and Dr. Iain Wallace for taking time off their busy schedules for reviewing my work and for providing valuable feedback.

I would also like to thank my industry partner Ericsson Inc. for giving me the opportunity to perform a part of my research at Ericsson and for providing me the funds and resources. Many thanks to Ronald Casselman for pushing this industry-academia collaboration. I sincerely thank Dr Sigrid Eldh for her never-ending support and encouragement and for the time she spent reviewing my work.

To my manager at Ericsson, Johanna Nicoletta, 'Thank you' is too small a word to express my appreciation and gratitude for the support and guidance that you have showed me during my stay at Ericsson. Thank you for not thinking twice and providing me with all the resources required for my research. Many thanks to the Senior Specialists at Ericsson Ottawa, Geoff McHardy and Gordon Moore for all those valuable technical discussions contributing to the successful completion of my research project.

I humbly thank the administrative and technical staff of SCE at Carleton University. Your efforts spent at running everything smoothly never go unnoticed.

When things go slow, take a coffee break! My fellow researchers and colleagues at SQUALL lab, thanks for the motivating and enjoyable environment and for proving that an office without windows is not always without fun.

My perseverance and inspiration comes from my life-force; my parents and their continuous support and prayers. Thank you my dear parents for pulling me up and reminding me of the silver lining when all I saw were dark clouds.

Sincere words of thankfulness to my dear parents-in-law for their love and support and for the continuous reminder that nothing is impossible.

Thank you my dear brother Indran for your care and encouragement.

To my dear husband Ripudaman, thank you for putting up with my ups and downs, my failures and successes and my nerve-racking and long days at work.

Last but by no means the least, a million thanks to all my family and friends for constantly buzzing me when I was barely visible and for reminding me of the wonderful people in my life.

My heartfelt gratitude to all!

- Sunint K. Khalsa

Table of Contents

List of Tables

List of Illustrations

Chapter 1 ---Introduction

Software testing is the process of ensuring a software under test (SUT) performs as intended. Software testing techniques can be broadly categorized as black box, when test case construction focuses on functionality, or white box, when test case construction uses the internal logic and structure of the code. One way to perform black box testing is to identify the input domain of the SUT leading to an effective testing of the system. This process of defining the characteristics of inputs and outputs parameters [1] with precision is called input parameter modeling. Among the various techniques for input parameter modeling (e.g., Classification Trees [2], Anti-random technique [3]), Category Partition (CP) is a black box testing technique proposed by Ostrand and Balcer [4], which incorporates both Equivalence Partitioning and Boundary Value Analyses [5]. CP provides a high level and concise way to represent the test cases. This technique helps to organise the testing activity by making it more systematic and understandable while assuring that the important test input combinations are not overlooked. In this dissertation we will primarily talk about Category Partition technique [4].

1.1 Category partition implementation process

Figure 1-1 shows the process followed for the implementation of the Category Partition (CP) technique. The implementation of CP begins by identifying the parameters and environment variables of a functionality to be tested. We first note that CP scales up since this functionality can be implemented by a function or method in a class, a component, an entire software system. In the rest of this document, we will simply use System Under Test (SUT) to refer to any implemented functionality that is tested with

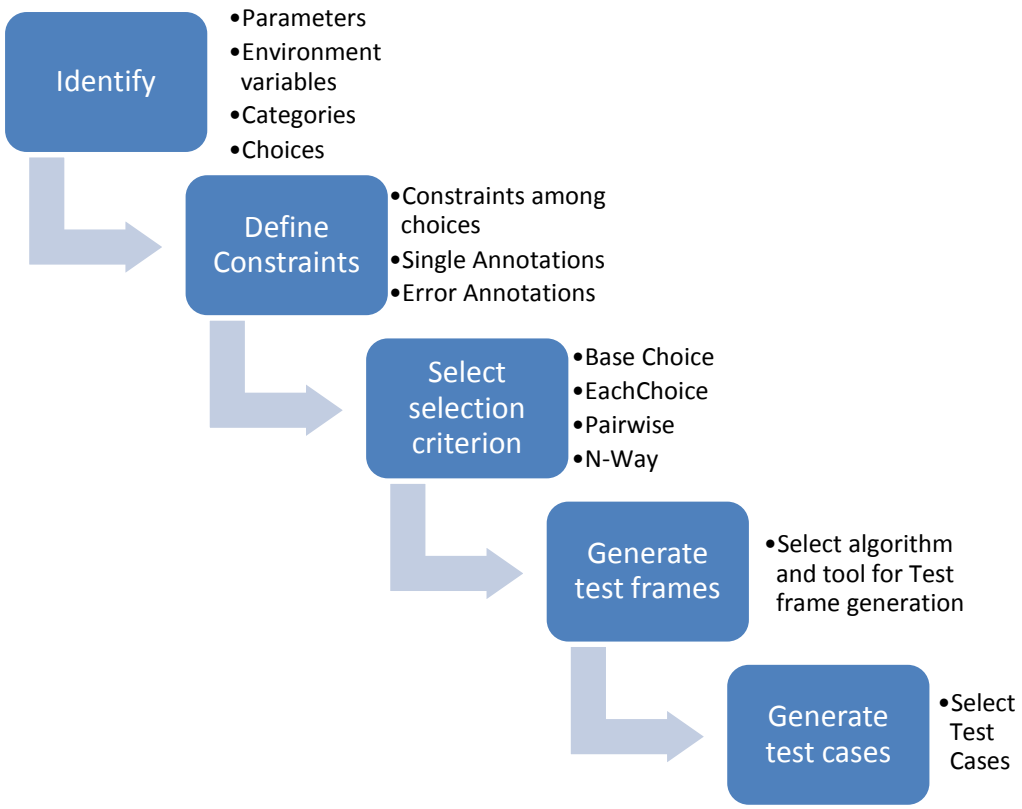


Figure 1-1: Category partition implementation process

CP. Parameters are the inputs to the SUT either by the user or some other functional unit. Environment variables are aspects of the execution environment that may influence the behaviour of the SUT. These parameters and environment variables are then envisioned into categories, which are characteristics of the parameter (or environment variable) that are deemed important from a testing point of view. Each characteristic leads to the definition of so-called choices, which are equivalent classes, possibly using boundary value analysis, splitting the domain of values (implicitly) defined by the characteristic. The choices should be disjoint and together they should represent the whole range of values for that specific category.

In the second step, various types of constraints are introduced. Constraints are used to specify for instance that some choices from two different categories should always be used together, can never be used together, or can be used together under certain conditions. Constraints such as Error and Single annotations are used to limit the occurrence of a few selected choices and thereby reduce redundant combinations. These special "error" and "single" constraints specify that a choice can only be used once in the combination of choices called 'test frames'. An error choice indicates that the test script should expect an exception or an error state, whereas single indicates a condition (choice) that only needs to be exercised once (according to the tester's expertise). In this thesis, the parameters, categories, choices and the constraints on the choices together with single and error annotations is collectively called the category partition specification of the SUT.

The choices (at most one choice per category) are then combined thanks to a specific selection criterion, such as Base Choice, Each Choice or Pairwise (step 3). Each Choice ensures that every choice of a category is included in at least one test frame [6]. The Base Choice criterion lets a tester select a base (important) choice in a category. The first test frame combines all the base choices and other test frames are created by holding all but one base choice constant and using each non-base choice once for the one non-constant choice [6]. The Pairwise criterion ensures that all possible pairs of choices from two different categories are a part of the test suite [1]. A test engineer can decide on a suitable criterion on the basis of resources available for testing. Once the criterion is chosen, test frames are generated. If the SUT is large, consisting of a large number of categories and choices, a manual generation of test frames will not be very efficient. Therefore, a test

engineer can rely on a test frame generation algorithm or tool for the generation of test frames (step 4), using combinatorial testing techniques for instance. These techniques can generate test frames in the presence or in the absence of constraints [7]. The obtained test frames, which are essentially test case specifications, are provided actual values, in step 5, to produce test cases: one test frame typically becomes one test case, although it is possible (though costly) to identify several sets of test inputs and therefore several test cases for a single test frame.

For a large system, using CP can lead to a large number of categories and hence choices. The combinations of all these choices can further lead to a large number of test frames. These test frames along with large numbers of test inputs, can result in a large number of test cases. There is then a risk that the resulting test suite will violate the property of Finite Applicability [8], which asserts the creation of finite test cases in order to test a given program. Hence, it becomes important to strategically select choices in the categories, strategically assign test inputs to those choices such that the inputs are representative of the whole partition and their combinations result in finite test cases leading to uncovering a maximum number of faults. Additionally, the use of selection criteria [1, 7] and algorithm used for the generation of test frames may have an impact on the final test suite quality.

The Category Partition technique, besides being more systematic than boundary value analysis and equivalence partitioning, lacks proper tool support. There has been traces of partial automation by few research groups. The Test Specification Language (TSL) [4], which was originally created for CP, is a textual language which does not facilitate automation, although it was used to semi-automatically generate test scripts [9]

(input values were selected manually). Briand et. al. [10] and Labiche et. al. [11] automated the process of generation of test frames from a CP specification with a tool support called Melba. Melba supports the definition of parameters, categories and choices and facilitates the specification of constraints among the choices in a Conjunctive Normal Form. Further Melba with its integration with CASA can generate test frames using the Each Choice, Pairwise and Base choice selection criteria.

1.2 Decisions involved in an effective CP implementation

In this section, we will follow the steps used for the generation of a CP specification for an example case study and will demonstrate various decisions that we will make for the generation of a test suite. We will also pay attention to what other decisions we could have made and which could have affected the outcome and hence the quality of the test suite. We will demonstrate the process using a simple example; a CP specification consisting of two parameters.

1.2.1 Identification of parameters, categories and choices

Assume a simple functionality that searches for the location of the first occurrence of a character in a string. As a first step (recall Figure 1-1), we identify two input parameters for this system: a string and a character. This is followed by the identification of characteristics (categories) and choices for the categories. For the first parameter (string), we identify the following categories: Length of string, number of occurrences (of character in string), position of first occurrence and valid character string (given that the string should only have valid characters). For the second parameter (character), we identify one category: valid character. Further, for each category, choices are envisioned; for instance as shown in Table 1-I, one may consider “0”, “1 to max” and “>max” as

choices in the first category or can consider yes and no as two choices for the 'Valid Character String' category.

A tester here can decide on any number of categories or choices. For instance, a tester can decide not to include valid character string as one of the categories or length "0" or ">max" as the choices in category "Length of string". This decision of the tester, i.e., whether the tester needs robustness testing or not depends on what the tester believes is important from a testing point of view. For instance, this functionality can be a part of some larger system where such invalid inputs are not expected. In that case, a tester can decide not to include choices corresponding to robustness testing, which can reduce the size of the CP specification and hence the number of test cases in the test suite.

Furthermore, we will decide over the base choice for each category. A base choice is an important or critical choice in the category that needs to be tested more often than the other choices and its selection is based on domain knowledge. We selected choices 1.2, 2.3, 3.2, 4.1, and 5.1 as base choices (Table 1-I).

Table 1-I: Example CP specification

Parameter	**Category**	**Choices**	**Constraints**
String	1. Length of String	[1.1] 0	[Single] if (2.1)
		[1.2] 1-max	**Base choice**
		[1.3] >max	[Error]
	2. No. of occurrences of character	[2.1] Zero	If (3.4)
		[2.2] One	If !(3.4)
		[2.3] Many	**Base choice** If !(3.4)
	3. Position of first occurrence	[3.1]First	
		[3.2] Middle	**Base choice**
		[3.3] Last	If (2.2)
		[3.4] Null	
	4. Valid character string	[4.1] Yes	**Base choice**
		[4.2] No	[Error]
Character	5.Valid character	[5.1] Yes	**Base choice**
		[5.2] No	[Error]

1.2.2 Identification of constraints in the CP specification

In the next phase, we implement constraints in the CP specification. We annotated choices as single and error to ensure the resulting test suite has only one test frame involving those choices (Table 1-I). Further, we also introduced constraints among the choices to ensure only valid test frames, i.e., choice combinations, are obtained. For instance, we introduce the constraint between choice 2.2, specifying that the character to be searched appears only once in the string, and choice 3.4, specifying that the character to be searched is not in the string, to prevent their combination: the constraint is "If !(3.4)". A tester can decide not to make single and error annotations in a CP specifications. The tester can also be cautious about the cost of the final test suite, which can affect his decisions on constraint selections. The tester might also have to decide about the constraint representation technique depending on the algorithm or tool which will be used for the generation of test frames while satisfying the constraints. This step however can be automated as in Melba [11].

Assume a CP specification with a large number of categories and choices. Introducing all types of constraints (error, single and constraints among choices) in such a CP specification can be resource intensive. On the other hand, the test engineer can also decide over including single and error annotations and omitting detailed constraints among choices. Unfortunately until now, there is no study that shows the impact of making decisions on different types of constraints and therefore assisting a tester on the impact of such decisions. These decisions are important for the tester and are taken into account in case the resources are limited.

1.2.3 Identification of selection criterion

Once the CP specification is obtained (Table 1-I), choices are combined based on a selection criterion. The tester decides which criterion will be most effective based on cost and fault detection effectiveness, i.e., the resulting test suite is expected to discover a maximum number of faults with a minimum number of test cases. The criteria can be Base Choice, Each Choice, Pairwise or All Combinations [6].

A Base Choice criterion begins by selecting a base choice for each category. A base choice can be the most important, most likely used or the most often used choice in the category. The selection of the base choice depends on the expertise of the test engineer. A base test frame is then created by combining the base choice for each category. Other test frames are created from the base test frame by replacing each base choice, one choice at a time, with a non-base choice of the category. This results in using each non-base choice at least once and each base choice several times. An Each Choice criterion ensures that every choice of a category is involved in at least one test frame. All Combinations or N-wise criterion ensures that all the possible combinations of choices are included in the test frames. Ostrand and Balcer [4] used this criterion while introducing the category partition technique. This criterion is often used as a benchmark for studying other criteria with respect to cost since it gives the upper bound, the largest set of test frames that can be created. The Pair wise criterion ensures that each choice of each category is paired with every choice of every other category [12].

If we consider the Base Choice criterion, the first test frame will be the choice combination consisting of all the base choices (1.2, 2.3, 3.2, 4.1, 5.1). The subsequent test frames are obtained by changing each base choice to a non-base choice, one at a time. We

begin with the first category and change the base choice 1.2 to non-base choice 1.1: the test frame is then (1.1, 2.3, 3.2, 4.1, 5.1). Since 1.1 is constrained with 2.1, we will have to change 2.3 to 2.1: the test frame becomes (1.1, 2.1, 3.2, 4.1, 5.1). Further, since 2.1 is constrained with 3.4 it will not be possible to pursue with the current base choice of category 3. Given the current definition of the base choice criterion and its insufficient support to handle complex constraints, it will not be possible to obtain the test frame with all these non-base choice together [6, 11]. This results in no test frame with choice 1.1 arising from the inability to satisfy constraints. The second test frame, resulting from the change from 2.3 to 2.1 will result in: (1.2, 2.1, 3.4, 4.1, 5.1); after 3.2 is changed to 3.4 due to the constraint. The third test frame is (1.2, 2.2, 3.2, 4.1, 5.1). The fourth test frame results from changing 3.2 to 3.1: (1.2, 2.3, 3.1, 4.1, 5.1). The fifth test frame is the result of the change from 3.2 to 3.3: (1.2, 2.2, 3.3, 4.1, 5.1); after satisfying the constraint. The next test frame is obtained by changing 3.2 to 3.4. This would lead to changing 2.3 to 2.1 due to constraint and the test frame will be 1.2, 2.1, 3.4, 4.1, 5.1. Once the base choice adequate test frames are obtained, the three error choices can be used to make three test frames using the Single Error coverage Criterion [13]. This criterion ensures that an error choice of each category is combined with non-error choices of other categories exactly once to form a test frame.

Analyzing the test frames obtained using the Base Choice criterion we observe that the test suite is devoid of choice 1.1, resulting in a test suite which is not Each Choice adequate. Faezeh et al. [11] also obtained these results using the Base Choice criterion. Therefore, if a tester decides to use the Base Choice criterion for a complex CP specification, it may not be possible to obtain an Each Choice adequate test suite as the

criterion does not currently support complex constraints, i.e., constraints with a lot of conjunctions and disjunctions. Therefore, the tester might have to revert to combinatorial criteria, e.g., pairwise or 3-way, because of abundance of tools providing constraint support. This however has a cost.

1.2.4 Identification of an appropriate tool for test frame generation

If the test engineer decides to generate test frames based on the Each choice or Pairwise criteria, the next decision will be on the selection of a strategy, algorithm, or tool, depending on the level of automation required, that will be used for the generation of test frames based on the selection criterion. Many combinatorial testing tools and algorithms are available; they use different algorithms for the generation of test frames, e.g., metaheuristic, greedy technique. Each technique uses a different strategy for the generation of test frames that can result in different combinations of choices and can likely affect the quality of the resulting test suite. There is a lack of compiled study on the suitability of tools for the Category Partition technique. For instance, there is no compiled study which can help a tester by precisely answering questions like: Which tool supports Base Choice? Which algorithm is deterministic? Which tool supports constraints and mixed covering arrays (where each category can have different number of choices) or is better in cost and effective at finding faults? In other words, there is a lack of study to assist the tester in deciding which tool or algorithm is most befitting to their needs. It is important to note that until now, there is no tool that supports constraints when the criterion used for the combination of choices is Base Choice.

1.2.5 Identification of test inputs for generating test cases

Once a tester decides on an algorithm to automatically generate the combination of choices, called test frames, the next step is the decision of the generation of test cases from test frames (decision 5). A test frame is a test case specification that needs to be concretized with actual test input values to become a test case that can eventually be implemented in a test script to be executed. For the Base Choice test frame mentioned above, (1.2, 2.3, 3.2, 4.1, 5.1), which specifies valid inputs (string and character), a string of nominal size with several occurrences of the searched character and the first occurrence of the character being somewhere in the middle, one possible test case will be string= “abscscb” and character = “c”. Instead of making manual selection of test inputs, a tester can decide to select the test inputs automatically. Different technologies, e.g., constraint solvers or theorem provers, can be used for this purpose.

1.3 On decisions to be made when applying CP—a summary

All the five decision points mentioned above, compiled in Figure 1-2, may each have important impacts on the quality of the resulting test suite by affecting its cost and effectiveness at finding faults. For instance, fewer choices means fewer tests (cheaper) but also higher risks to miss faults; introducing constraints can lead to valid test cases but can also increase the cost; different selection criteria can capture different types of faults and can result in different cost and fault detection effectiveness; different technologies used to create test frames and identify test inputs can result in different costs and effectiveness at finding faults. Considering all these variation points, it is paramount to obtain assertive conclusions based on experimental evaluations.

This thesis is an attempt at answering some of these questions. It starts with a discussion of the specific problems being addressed in the thesis and its contributions (Chapter 2). Chapter 3 discusses background and related work. Chapter 4, presents an orchestrated survey on the available algorithms and tools to support the automated construction of test frames. Chapter 5 defines the Category Partition specification of academic and industrial case studies. Chapter 6 discusses the results of using CP on the industrial case studies. Chapter 7 discusses the experimental evaluation of the impact of different types of constraints in CP. Chapter 8 introduces two different flavours of the Base Choice criterion to better support constraints. The performance of the new criteria and their comparison with existing ones is presented in Chapter 9. Chapter 10 discusses threats to validity and we conclude the thesis in Chapter 11.

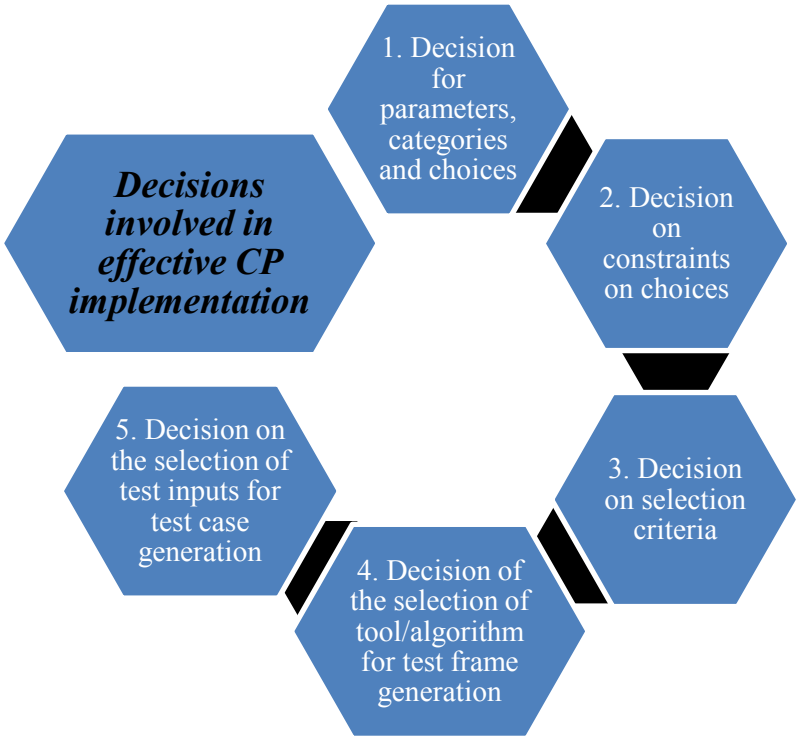


Figure 1-2: Different decisions made while implementing Category Partition technique

Chapter 2 —Problem Addressed by the Thesis and Contributions

While implementing the CP technique the tester has to undergo a series of decisions (Figure 1-2) in order to obtain an effective test suite. This thesis aims to assist a tester in making such decisions. This thesis addresses problems related to decision 2, 3 and 4 in Figure 1-2 and contributes in solving those problems.

2.1 Problems addressed by the thesis

As proposed by Ostrand and Balcer [4], "contradictory" test frames produced with CP can be removed by determining constraints among the choices. Additionally, the choices can also be annotated with "error" or "single" to combat redundancy [14]. Error and single annotation can be assumed as a type of constraint on the choices to ensure that the annotated choices do not appear more than once. Therefore, constraint among choices, error annotations and single annotations are three types of constraints, which ensure that there are no infeasible and redundant test frames. We could not find any study that shows the impact (for instance on fault detection, code coverage or cost) of using these three types of constraints against not using them at all. Although it is interesting to note that combining choices without accounting for constraints may result into infeasible test frames, i.e., test frames for which it is not feasible to find inputs [11], **there is no study to gauge the importance of each constraint type which can help their prioritization in case of limited resources. We therefore consider this as our first research problem**. Such a lack of study makes it hard for a tester to make decision 2 (Figure 1-2) while implementing the CP technique. Further, the need of this study arises

from the fact that constraint satisfiability is a resource intensive process. Such a study will be handy and will help testers decide how much constraints (error, single, constraints among choices) are enough.

There are four main selection criteria specific to the Category Partition technique [1]: Each Choice, Pairwise, All Combination and Base Choice. The first three criteria are akin to n-way testing and can be supported by Combinatorial Testing (CT) tools while accounting for constraints among choices. CT tools do not intrinsically support the Base Choice criterion. This criterion on the other hand relies on base choices selected by the tester and their combinations whilst satisfying constraints. The constraint handling in the base choice criterion so far deals with simple constraints [4]. It is interesting to note that when there are constraints on choices, this definition of Base Choice does not necessarily ensure that each choice is exercised in the set of test frames [11]. Moreover, there is very limited literature on how to handle constraints among choices while using this criterion. Amman and Offutt [6] talk about infeasible combinations of two choices but there is no mention of how to handle more complex situations. In other words, because of constraints on choices, Base Choice does not subsume Each Choice, which is somewhat counter-intuitive. There is a lack of literature on how a system with highly constrained inputs can implement the Base Choice criterion. Therefore**, the second problem, which the thesis will address, is the extension of the Base Choice criterion to support constraints among choices**. While extending the Base Choice criterion and comparing its results with other criteria we will help a test engineer by providing them experimental results to facilitate decision making, specifically decisions 3 (Figure 1-2).

After deciding over the selection criterion, a test engineer needs to decide over an appropriate Combinatorial Testing (CT) tool for the generation of test frames. This decision might have an impact on the quality of test suite. We tried to search for an appropriate tool but there was no compiled study to help us make this decision. Although CT is a strong area of research, there is limited compiled literature on using CT for the purpose of CP, i.e., selecting a CT tool befitting to the needs of a CP specification. **Therefore, the third problem that will be addressed by this thesis is the systematic characterization of combinatorial testing tools and algorithms specifically for CP testing that can help a test engineer make decision 4** (Figure 1-2).

The last problem that this thesis will address is the use of CP for real industrial case studies related to the field of telecommunication. Before a tester gets into the intricacies of implementing CP, an important question always stands: Is the implementation worth the effort? **Addressing this problem by implementing CP on the software units from the telecom domain can help a test engineer gauge the suitability of Category Partition to this domain**.

2.2 Contribution of the thesis

The dissertation contributes the following while addressing the problems defined in section 2.1.

In order to address the **first problem** we conducted a detailed experimental evaluation of the impact of different types of constraints on academic and industrial case studies. We used two academic case studies and three industrial case studies, designed 12 experiments and generated test frames using two combinatorial testing tools (CASA and ACTS) to gauge the impact of each constraint when the other two constraints are present

or absent. **We conclude that using all three types of constraints always results in a high quality test suite**. **However, if the test engineer has limited resources and decides to use only one constraint type, Error annotations are most effective**. **If the test engineer decides to use a combination of two constraints then Error and Single annotations give maximum return**. We obtained similar results for both tools. Further, the thesis also contributes on how these constraints impact the cost of the test suite. As mentioned by Cohen et. al. [15], introducing constraints does not always result in lowering the cost of the generated test suite. In case of complex constraints, introducing constraints among choices, single or error annotations can results in an increased cost because more combinations, and therefore more test cases, are required to generate an adequate test suite. **Another contribution of this thesis is the suggestion of threshold values, on the basis of quantitative analysis of experimental results, beyond which the cost of the test suites increases when Single, Error and Constraints among choices are used.** We obtained the values for both academic and industrial case studies.

This thesis also contributes two variants of the Base Choice criterion to better account for constraints, in an attempt to solve the second problem (section 2.1). The two variants are Constrained Base Choice (CBC) and Extended Constrained Base Choice (ECBC). Contrary to the Each Choice and Pair-Wise criteria, which, similarly to many other selection criteria (e.g., structural ones), have a descriptive definition, the Base Choice criterion has a prescriptive definition [6]. The former criteria tell us which test objective to achieve without telling us how to do it whereas the latter criterion tells us how to proceed to obtain a set of adequate test frames. Our extension to Base Choice, i.e., CBC and ECBC, also follows a prescriptive approach. We will argue that, by

construction, CBC subsumes BC and ECBC subsumes CBC. **Further, the results that we obtained with the new criteria have test frames from Base Choice criterion and traces of frames from Pairwise and 3-way leading to results equivalent to a 3-way criterion but with a much lesser cost.** We can relate our results with those of Grindal et. al. [13] where the combination of Base Choice and Pairwise criteria gave better results than the individual criterion, hence proving that the criterion obtained resulting from the combination of two or more criterion is more effective.

The problem of generating test frames using CP is akin to the generation of Covering Arrays using combinatorial testing tools. A Covering Array is an array of rows and columns where the columns represent parameters and the rows represent the test cases. In order to facilitate our research, we searched for a suitable Covering Array Generation Tool. However, with a large number of commercial and academic tools and algorithms and no systematic characterization specifically for Category Partition technique, we could not narrow down to a few most suitable tools for our purpose. **Therefore, we performed a survey [16] on the basis of various research questions and published a list of available algorithms and tools and their categorization on the basis of these research questions**. We identified 75 of such tools and categorized them based on technique and test generation strategy used by each tool or algorithm for the generation of covering arrays. We also categorized the tools on the basis of the selection criteria supported by each tool/algorithm, coverage strength support, constraint support and mixed covering array support. We did not choose cost or time taken by a covering array algorithm for the generation of covering array as a criterion for tool categorization because there are many survey papers in the domain of mathematics or computer science which solely talk about

this aspect of CA tools and algorithms [17, 18]. The survey has tool and algorithms available until 2014 as that was the time when we were looking for an appropriate tool to support the automation of some aspects of our research work. Researchers, practitioners or testers who wish to select one of those tools befitting to their needs can use the results of this survey, possibly extending it to 2017 thanks to our disclosure of our survey protocol. This survey helped us solve the **third problem** mentioned in section 2.1 and we could select a tool befitting to our needs. **We continued further by performing experiments using the tools selected from our survey and generated test frames using three of such tools on our 13 academic and industrial case studies and using different selection criteria. The thesis contributes by discussing the results from these experiments and concludes on the effect of test frame generation technique on the quality of test suite.**

The thesis also contributes by implementing Category Partition Testing on industrial case studies in an attempt to solve the fourth problem (section 2.1). This study helped us in accessing the suitability of Category Partition testing on a telecommunication software. This implementation resulted in better results than when a systematic technique for input parameter modelling was not used. We followed the implementation with quantitative analysis of the results.

The thesis also contributes a large number of experiments, conducted using different case studies (academic and industrial), using different CP specifications, defining constraints for different combinatorial testing tools, selecting appropriate combinatorial testing tools for the generation of test frames, using various selection criteria and describing the results quantitatively or qualitatively. For the purpose of

these experiments, we use three academic case studies and ten industrial cases studies. The case studies vary in terms of size of the CP specification and number and complexity of constraints. We compare the results of the proposed criteria (CBC, ECBC) and existing criteria (Each Choice, Pairwise, 3-way combinations, Base Choice) using the ACTS [19], CASA [20] and PICT [21] combinatorial testing tools for the generation of test frames. We conduct primarily a quantitative analysis of cost (i.e., number of test frames) and effectiveness at covering the source code, data flow and detecting faults, which we complemented with a qualitative analysis.

2.3 Publications

This thesis resulted in the following publications:

- S.K. Khalsa and Y. Labiche, *An orchestrated survey on available algorithms and tools for Combinatorial Testing*, in 25th IEEE International Symposium on Software Reliability Engineering (ISSRE). 2014. p. 323-334.
- S.K. Khalsa, Y. Labiche, and J. Nicoletta. *The power of Single and Error annotations in Category Partition Testing: An Experimental Evaluation,* in 20th ACM International Conference on Evaluation and Assessment in Software Engineering (EASE). 2016. Article no. 28
- S.K. Khalsa, Y. Labiche. *An extension of Category Partition testing for highly constrained systems,* in IEEE 17th High Assurance Systems Engineering Symposium (HASE). 2016. p. 47-54
- S.K. Khalsa, Y. Labiche. *Extending Category Partition's Base Choice criterion to better support constraints*. Journal of Software: Evolution and Process (Wiley). *DOI: 10.1002/smr.1868.* April 2017.

Chapter 3 —Background and Related Work

Identifying input parameters is integral to black box testing. There are various techniques for input parameter modeling such as Classification Trees proposed by Grochtmann et al. [2] which are based on Boundary Value Analysis and Equivalence Partitioning and is defined in a tree like structure. The anti-Random technique proposed by Malaiya [3] is based on the selection of test inputs on the basis of the hamming distance [22]. Malaiya propose an approach to translate the input space into a binary representation and calculate the hamming distance between two binary vectors. A greater hamming distance symbolizes diverse test cases. Chen and colleagues suggested a UML activity diagram based method [23] as well as a choice relation framework [24] where the latter is extended from classification trees [2]. Chen et al. [25] compared the choice relation framework and the classification tree method but without a discussion on constraints on choices. Elmendorf [26] proposed a technique for defining causes of a function's behaviour and the effects of those causes and named the technique Cause Effect Graphs. Grindal et. al [27] proposed a model for input parameter modeling specifically for performing combinatorial testing. This model is similar to Category Partitioning and provides a detailed strategy, checklist and guidance for selecting the input parameters.

Myers [5] proposed the technique of Equivalence Partitioning and Boundary Value Analysis for partitioning the input domain into groups and selecting a value from each group. The driving force for creating partitions of the input domain was the unmanageable number of inputs resulting from exhaustive testing. Boundary values

analysis selects the input values along the boundaries of the partitions that are exercised while testing the system. Although both the techniques are well known they lack a systematic integration and implementation. Category Partition proposed by Ostrand and Balcer [4] incorporates both Equivalence Partitioning and Boundary Value Analyses and follows an organized approach for the generation of tests by systematically integrating both techniques.

Category partition [4] has been used in many different contexts and level of granularity, ranging from functions to state machines to use cases [4, 9, 28-30], either manually or by using executable test scripts [9, 10]. We will discuss the related work within the framework of the decisions a tester has to make (recall Figure 1-2) for the implementation of Category Partition. Section 3.1 discusses issues with respect to the definition of categories and choices (decision 1). Section 3.2 discusses the related work about the specification of constraints on choices (decision 2). Section 3.3 discusses the related work about selection criteria (decisions 3). Section 3.4 discusses the related work about tool support for the generation of adequate test frames (decision 4).

Our contribution in this thesis is to decisions 2, 3, and 4 (Figure 1-2), so we have introductory literature, instead of a detailed one, for decision 1 and decision 5.

3.1 On the definition of categories and choices (decision 1, Figure 1-2).

A research group in Hong-Kong led by P. L. Poon has been actively researching CP. They identified patterns of categories and choices that can be defined once and reused [31]. They also defined a divide and conquer methodology for identifying categories and choices which they extended from a choice relation framework [24], and named it DESSERT [32]. They further have empirical studies to their credit where they

demonstrated the mistakes test engineers usually make while implementing CP [33]. There is also an interesting work that shows the impact of user expertise on the outcome of CP [34]. They evaluated the effectiveness of a checklist which they provided previously [33] for detecting the missing categories or choices. The work of this research group mainly focuses on defining the CP specification i.e., categories and choices accurately and precisely, and is complementary to ours for the overall use of CP.

3.2 On the definition of constraints for choices (decision 2, Figure 1-2)

Chen et al. [24] developed a choice relation framework for supporting test case generation using category partitioning. They suggested means to systematically capture constraints among choices and generated a choice priority table. Grochtmann et al. [2] proposed a Classification Tree method for building a model of constraints for a given input domain to facilitate the construction of such specifications. Grindal et al. [35] investigated various conflict handling strategies in their work.

The research work in category partition testing intermittently talk about constraints among choices. For instance, Offutt et al. [6], Poon et al. [14, 36] mention constraints in their work but do not discuss the handling of complex constraints neither do they discuss the impact of these constraints on the quality and cost of the test suites. Since the software systems nowadays are highly complex, resulting in complex constraints among the input parameters, there is a lack of concrete literature on constraint handling in Category Partition testing. Because of this lack and because of the fact that generation of test frames using CP for the Each Choice and Pairwise selection criteria is akin to 1-way and 2-way combinatorial testing [11], we look at the constraint satisfiability in the combinatorial testing techniques [37-41].

Justyna [42] and Colbourn et. al. [40, 42] in their work underlined the importance of constraints in combinatorial testing and categorized the constraints as hard and soft constraints. A hard constraint is a constraint which, if not satisfied, will result in at least one infeasible test frame (i.e., a test frame for which one cannot find test inputs that satisfy the choice specification), and therefore insufficient testing in (the likely) case that such infeasible test frames are simply dropped. A soft constraint is optional: specifying the constraint or not does not impact feasibility of test frames; A test engineer may decide to specify such constraints according to her knowledge of the problem domain, for instance to reduce the number of test cases and therefore testing costs. Having constraints can also result in more combinations of non-constrained choices, which improves the quality of the test suite as proved by our experimental data in Chapter 7, an observation in line with the assumptions made by Danziger et. al. [43]. They assumed that a pair of choices that has been visited once will not be visited again, but their study is based on constraints expressed as forbidden tuples and is limited to pairs. Our results are based on case studies that have constraints which span more than two choices. Alternatively, not having soft constraints (e.g., error ones) can also lead to extended robustness testing since this results in more choice combinations with the otherwise annotated error choices, particularly in programs that have good defensive programming code.

The constraints that are supplied to a combinatorial testing tool are represented using a specific representation style that is acceptable to the tool. There can be different representations for constraints: Forbidden tuple representation or full representation to name a few. In case of forbidden tuples, the forbidden combinations are specified as constraints. A SAT solver or greedy algorithm then ensures that those combinations

never occur. Constraints can also be specified using first order logic formula which is a representation used by a combinatorial testing tool called ACTS [19]. The logic formula can use Boolean, relational or arithmetic operator. A recent work by Cohen et. al. [44] empirically analyzed that specifying constraints as forbidden tuples yield faster results than with other methods of specifying constraints. Their results are based on experiments done with ACTS and CASA [20].

Cohen et al. [15] in their work surveyed various constraint handling techniques which can be used with existing combinatorial interaction testing tools. They mentioned, in particular based on examples, that constraints among parameter values do not always reduce the size of the test suite. However, there is no mention of how a test engineer can use this information and decide on a tradeoff between how much constraints to define and how will that affect the size of the test suite.

There is an interesting work by Poon et al. [36] which they further extended and evaluated [14] where the authors empirically compare test-once strategy and test-a-few strategy. The authors employ these strategies on particular conditions in the specifications and they test those conditions either once (test-once strategy) or a few times (test-a-few strategy). They mentioned that the success of the test-once strategy is based on the 'uniformity assumption', i.e., an implementation should process the condition uniformly irrespective of values used for the testing of these conditions. We are assuming the test-once strategy to be similar to annotating the choices as Single and Error in a CP specification. The driving force while selecting a choice as Single is that such a choice should behave uniformly irrespective of test inputs. Because of this uniformity, only one occurrence of these 'Single' or 'Error' annotated choices are sufficient for adequate

testing because additional test cases will not contribute to additional coverage but to cost. In Poon's paper, authors claim that such an annotation is useful in reducing the cost of the test suite. However, if there is an error in the implementation of the condition, this would violate the uniformity assumption and we would need more than one test case with different test inputs to uncover the faults. Similarly, an incorrect Single annotation can fail in discovering certain faults because of its test-once behaviour. The test-a-few strategy on the other hand, exercises choices several times in the test suite which obviously leads to better quality test frames than the test-once strategy, as is also proven by the authors. They further discussed three strategies based on test-a-few and discuss the fault detection capability of each strategy. Besides being an interesting contribution there is no mention of the selection criterion used for the generation of test frames using the test-once and test-a-few strategies.

Danziger et. al. in [43] in their research proposed a theoretical framework to reduce the size of the covering arrays that employ constraints. The constraints are set up as forbidden pairwise interactions and they did not discuss three-way or N-way interaction of forbidden tuples. In the present work, we consider constraints that span beyond pairs. We prove in the present work (Chapter 7) that the introduction of constraints (constraint among choices, single and error annotations) does not always lead to a reduction in the cost of the test suite, for certain selection criterion and that cost depends of various factors.

To sum up, there is ample research on CP and its improvement in terms of identification of categories, choices (see previous section) and constraints between choices to obtain valid test frames. To the best of our knowledge, there is however no

study which discusses the impact of Single and Error annotations on the quality of the test suite. The closest study to our work is by Poon et. al. [14, 36], although they did not mention the criterion used for the combination of choices in test suites and their claim of a general reduction in the cost, in the presence of constraints is also not in agreement with our study. For the constraints that are implemented among choices (based on combinatorial testing, because of the lack of related literature in the field of category partition), Cohen et. al. [15] in their work mentioned that there can be an increase in the size of the test suite but there is no mention on how this information can guide a test engineer in deciding over the amount/complexity of constraints. Considering these three types of constraints (constraint among choices, error and single annotations) and the short time to delivery of the software product, we could not find any literature on the impact of three types of constraints on the test suite quality. This information can be useful for the test engineer in prioritizing certain constraints in case of limited resources therefore helping in gauging how much constraint are enough to make a software product readily available for delivery. Therefore, our work differs from, and complement related work in that instead of suggesting improvements in the CP technique, we conduct and report on experiments in which we studied the cost and effectiveness of implementing different types of constraints in the CP specification. We studied each constraint in the presence and in absence of the remaining two constraints and answer based on our results, questions like, which is the most effective constraint?

3.3 On selection criteria and their support for constraints (decision 3, Figure 1-2)

As discussed, there are various criteria for the selection of choices in Category Partition in order to form adequate test frames and hence test cases [1, 7]. The criteria

which are most commonly associated with the Category Partition technique are Each Choice, Base Choice and All combinations which were introduced by Amman et al. [6] and Pairwise [1]. Some variants are also proposed to the Base Choice criterion. Cohen et. al. [45] proposed the notion of a default value in place of base choices and suggested that a test case should always have a default value while at the same time changing other values. Burr et al. [46] proposed to have maximum and minimum values in the test case along with the default value. Nie et. al. [47] renamed the Base Choice Criterion as One Factor One Time (OFOT) while keeping the process of generation of test frames the same.

Grindal et. al [13] made some general observations regarding the fault finding ability of the combination strategies. They observed that, in their experiments, the Each Choice criterion surprisingly supplied good results, which they cannot always guarantee. Grindal mentioned that the Base Choice criterion uses domain knowledge for the identification of base choices; a reason why the Base Choice criterion detects different types of faults as compared to Pairwise. They also concluded that the Base Choice criterion performs better than the Pairwise criterion when there are few candidates for becoming a base choice in a category. When there is a large number of candidates for base choice selection, the Pairwise criterion gives better results. The authors also proposed that a combination of the Base Choice and Pairwise criteria can be more promising and can give better results. They experimented by combining these two criteria and observed that for some of the case studies the combination discovered more faults than each individual criterion. Their Pairwise criterion is implemented using two different techniques; Orthogonal Arrays [48]

which is based on mathematical object and Automated Efficient Test Generator which is a greedy algorithm [45].

Another criterion which is based on domain knowledge is the Input Output based parameter interaction [49]. Instead of exercising interactions of the complete set of parameters, the Input Output Based Interaction criterion splits the set of parameters into (possibly overlapping) subsets that each contain the parameters that impact the value of one output parameter [50]. Therefore, in order to implement this criterion it is important to know the behaviour of the functionality under test similar to the Base Choice criterion. A study performed by Othman et. al. [49] showed that Input Output based parameter interaction gives better results in terms of number of test cases and ability to find faults than Uniform and Variable Strength interaction [51]. Our intent to discuss the Input Output based parameter interaction is not for the sake of completeness but to emphasize the importance of a selection criterion which is based on domain knowledge.

If the SUT has a large number of categories and choices, it is infeasible to generate test frames manually. Therefore, the process of generation of test frames can be automated using test frame generation techniques which are based on Combinatorial Testing techniques. These techniques can be used to generate test frames for Each choice (1-way), Pairwise (2-way) or All combination (N-way) as the definition of All Combination by Ammann et. al. [6] is similar to N-wise coverage by Grindal et. al [7] which is based on combinatorial testing. However, there is a significant difference between Combinatorial Testing (CT) and Category Partitioning (CP). In CP the combination of choices are test frames which are in fact test case specifications. These test case specifications (test frames) are provided test inputs based on the choice

specifications to obtain test cases. Therefore, the final quality of the test suite will not solely depend on the test frames generation technique but will also depend on the strategy used for obtaining the test inputs on the basis of choice specifications. However, in the case of CT the choice specifications are in essence the values (choices) of the parameters (categories), therefore the execution of the combinatorial testing tool will generate test cases instead of test frames. There are studies in the field of combinatorial testing [52] which show that 100% fault detection can be achieved by up to 6-way combinatorial interaction testing. These studies also conclude that maximum faults are discovered with pair wise interaction (2-way) and single factor faults whereas comparatively fewer faults are discovered by interaction of three and more parameters. However, these studies and hence the results cannot be completely applied to the Category Partition technique because in the case of Category Partition the final fault detection will also depend on the test input selection strategy.

Since there is a lack of constraints handling tools in Category Partitioning we can use the constraint handling capability of combinatorial testing tools specifically while dealing with T-wise criteria (e.g., Each Choice, Pairwise) and complex constraints. These constraints help ensure that only valid test frames are obtained. A lot of literature is available on constraint handling in combinatorial testing [15, 20, 41, 53, 54] resulting in a number of tools and algorithms which support constraints during test frame generation [16]. Although there is ample research for constraint support in combinatorial testing, and therefore for Each Choice and Pairwise in the context of CP, e.g. Grindal et. al. [55], there is very limited literature on constraint handling for Base Choice in the context of CP.

Having discussed the importance of criterion based on domain knowledge and its ability to discover domain specific faults, we observe that these results are impacted when there are constraints among the choices as there are not enough guidelines on how these domain specific criteria can handle constraints. Faezeh and Labiche [11] observed that the presence of constraints between choices heavily impacts the use of those criteria, including Base Choice which, in some cases, does not exercise every single choice. Although Base Choice looks promising, there is a lack of literature on how to handle complex constraints, Ammann and Offutt [6] being an exception.

Ammann and Offutt [1, 6] proposed a method for handling infeasible combinations during the generation of test frames using Base Choice. If an infeasible combination is found in a test frame then they suggest to change a conflicting choice with some other choice of the category so that the test frame becomes a feasible combination. They illustrate their idea on a simple example containing two conflicting choices. We find that their definition of the procedure to handle infeasible test frames due to constraints is not precise enough; there is no guarantee that their solution would work on more complex situations involving constraints (e.g., when following their procedure on the example of Section 1.2 we could not obtain an each choice adequate test suite using the Base Choice criterion in the presence of constraints). This motivated us to improve the definition of the Base Choice criterion, which resulted in two different flavours of the criterion.

To summarize, there is no published work, to the best of our knowledge that can help someone deal with constraints on choices in a systematic way when producing a Base Choice adequate test suite in the context of Category Partition.

3.4 On techniques for test frame generation (decision 4, Figure 1-2)

Once a test engineer decides over the selection criterion to use, the next decision is the selection of an appropriate tool or algorithm for the generation of test frames. An engineer, based on the requirements of the Category Partition technique, can have different criteria for the selection of such a tool and can revert to literature to make a selection.

The research work by Kuliamin et. al. [17, 18] surveys methods for generating covering arrays. These papers survey covering arrays generation techniques on the basis of size and time of generation of covering arrays. They however do not discuss extensively the tools or algorithms which support a specific technique, the coverage strength or selection criteria. Other surveys (e.g., Grindal et. al. [7, 47, 56-58], Zamli et. al. [56, 57] and Anand [58]) focus on the techniques but do not discuss all the tools and algorithms supporting those techniques in detail. For instance, they do not discuss support for constraints or higher coverage strength, which is essential in our Category Partition context. The nearest work to our survey is by Rahman et al. [57], who discuss various techniques, their strengths and weaknesses along with the coverage strengths they support. They also mention if a specific technique supports constraints. They do not extensively mention the tools or algorithms supporting a specific technique, the constraint handling and representation technique adopted by a tool/algorithm, the selection criteria supported by a tool. In other words, we intend to provide a complete and compiled picture of what can be found in the literature to date. There is only one research work by Cohen et. al. [15], to the best of our knowledge, which discusses the constraint handling support in tools/algorithms for nine tools whereas we discuss 32 such tools (Chapter 4).

To summarize, none of the research work until March 2014, to the best of our knowledge, surveys the tools and algorithms as extensively as what we survey in this thesis or compare them on the basis of comparison criteria which we outlined in Chapter 4.

3.5 On techniques for the selection of test inputs (decision 5, Figure 1-2)

Once a test frame is obtained, the next decision is the selection of test inputs to form test cases. A test frame is a test case specification that is then concretized with actual test inputs to create a test case. Similar to the use of different technologies for the generation of test frames, different technologies can be used for the test input selection to obtain test cases. Test inputs can be derived manually using a documented procedure or it can be obtained automatically by feeding the test case specification to a Boolean Satisfiability (SAT) Solver or Satisfiability Modulo Theory (SMT) solver. These solvers decide over the logical satisfiability of the specification given a set of inputs. Different tools, e.g., Microsoft Z3 theorem prover [59], Yices SMT solver [60] can be used to find test inputs. Choosing such a tool can have an impact on the quality of the test suite. EvoSuite [61] is another tool that is based on evolutionary algorithms and is used to generate inputs for white box testing. Wegener et. al. [62] proposed an evolutionary test environment for generating test data for structural tests.

Chapter 4 —An Orchestrated Survey of Available Algorithms and Tools for Combinatorial Testing

In order to use the selection criteria when one has a large number of parameter (categories) and values (choices), a technique is required, which can effectively and efficiently make combinations. One such commonly used technique is Combinational Testing (CT). Combinatorial testing is the technique based on the mathematical concept of combinatorics, which is the study of combinations and permutations of sets of elements. This technique of combinatorics is widely used in performing black box testing of software systems. CT can be broadly applied at two levels; configuration level and input parameter level [52]. At the configuration level, system configurations are considered as parameters for testing, e.g., operating systems, browsers, network protocols etc. T-wise interactions among these configurations are then performed and covering arrays, a matrix of parameters and values, are generated. At the input parameter level, the actual inputs to the system or subsystem are considered. The inputs can be the actual values or partition of the input space as defined by equivalence partitioning. A related input parameter modeling technique for combinatorial testing has also been proposed by Grindal et. al [27] which is based on the category partitioning [4]. Yilmaz et. al [63] in their research work suggested a model consisting of four phases for performing combinatorial testing. They suggested modeling, sampling, testing and analysing starting from modeling the input parameters to analysing the results of the covering arrays.

The reader will notice we need two different terminologies. With Category Partition, parameters are characterized by categories, which are split into choices, and choices need

to be combined (one choice per category) to form test frames and hence test cases. In the CT domain, parameters have values and one combines those values (one value per parameter) to form test cases. We can establish a mapping between the two terminologies: categories and choices (Category Partition) map to parameters and values (CT). Unless otherwise specified, we will use the category/choice terms when the discussion is on Category Partition, and we will use the parameter/value terms when the discussion is on CT. We may need to mix terms, though without loss of clarity when one remembers the mapping.

4.1 Covering Arrays

Combinational Testing can lead to the generation of Orthogonal Arrays (OA) or Covering Arrays (CA). An Orthogonal array is a N x K array defined as $OA_\lambda(N; t, k, v)$ where N depicts the number of test frames (rows) and K defines the number of parameters (columns), t is the strength of combination and v is the number of values in the parameter. The value $\lambda = \frac{N}{v^t}$ defines the number of times each t-tuple exists in N x t subarray [64]. The OA requires every t-tuple to exist exactly λ number of times. However, this required is relieved in Covering Arrays where every t-tuple should exist atleast λ number of times. A CA is defined by $CA_\lambda(N; t, k, v)$ where N being the number of test frames, k is the number of parameters, v is the number of values in each parameters and t is the combinatorial strength. λ is usually assigned the value one (λ=1), which means that every t-tuple should exist atleast once in the N x t subarray [64]. When N is unknown or unspecified the notation CA(t, k, v) is often used. A CA is optimal if it contains minimum possible number of rows. Both OA and CA requires that all the parameters should have the same number of values. This condition is further relieved in

case of Mixed covering arrays (MCA) where each parameter can have different number of values. MCA where each parameter can have different number of values is represented by the notation MCA (N; t, k, C) where N is number of tests (rows), t is the combinatorial strength, k is the total number of parameters and C represents each parameter and its corresponding number of values, i.e., C= (v_1^{p1}, v_2^{p2}, ..v_n^{pn}). An example MCA can be defined as (25; 2, 21, 5^2, 3^6, 2^{13}) where the number of tests are 25, the combination strength is 2, total number of parameters are 21 where 2 parameters have 5 values, 6 parameters have 3 values and 13 parameters have 2 values each.

Variable strength covering arrays (VSCA) [47, 65, 66] are used when certain parameters are required to be tested more strongly than others, which mean that the coverage strength is higher for a subset of parameters. A VSCA is represented by VSCA (N; t, C, C_s) where N is the total number of tests, t is the combinatorial strength, C represents each parameter and its corresponding number of values combined with the strength t. Set C_s consists of a covering array of a subset of k parameters combined with a coverage strength greater than t, e.g., VSCA (12, 2, 3^2, 2^2, {MCA (3, 3^1, 3^2)}).

Constraint Covering Arrays (CCA) [15] are used to represent constraints along with the covering arrays in the form of forbidden tuples. They are represented as (N; t, k, v, F) where F corresponds to the set of forbidden interactions. The natural extension of CCA is MCCA, or Mixed Constraint Covering Arrays, represented as (N; t, k, C, F) where C= ($v_1^{p_1}$, $v_2^{p_2}$, ..$v_n^{p_n}$) and F is the set of forbidden tuples. e.g., CCA (N; 2, 3, 4, F) F= ((0, 5), (2, 11), (3, 7, 8), (2, 5)) (each of the three parameter have four values which are assigned unique number from 0 to 11 and the forbidden tuples are specified in F).

There are few differences in the features of OA and CA but for application in Category Partition, a CA is typically preferred for a number of reasons, including: an OA assumes that all the parameters have the same number of values, resulting in the same number of occurrences of the values in the test suite. This is rarely feasible in practice [7]. It is possible to construct a CA for any test specification i.e., with unequal number of parameters values, but an OA is not possible for such a specification. In case of constrained test specifications, it is feasible to generate CAs because each tuple is required to occur at least once in the test suite; this condition can be satisfied even in the presence of the constraints. However, OAs fulfilling this requirement can result in invalid combinations based on the constraints in the specification. Removing these invalid combinations explicitly will result in incomplete testing as some valid pairs might also be removed. Another advantage of CA is its extension to Mixed Covering Arrays. Because of these advantages, Covering Arrays are widely used with Combinatorial Testing (CT). There are a number of covering arrays discussed in literature Table 4-I. Yilmaz and colleagues [63] give a detailed picture of the types of covering arrays that can be used and the current research in the field.

In the context of Category Partition, a test engineer would be looking for a CA generation solution that could support one or more of the following: different categories typically have different number of choices; choices are typically associated with constraints to enforce or prevent some combinations or to ensure that a choice only appears once in the set of test frames; different selection criteria (e.g., Each Choice, Base Choice) can be considered to generate combinations of choices. Contrary to other studies that compare the effectiveness of CA generation technologies at producing the least

number of test cases in least amount of time, we are interested in functionalities of such technologies. Needless to say that it is always interesting to know the execution time or a size of an algorithm generating a covering and there are survey paper [17, 18, 64] and journal articles in domain of mathematics and computer science in support of that. In the current work, we mainly focus on the application of covering array is the domain of black box testing and we have not included the effectiveness of the CA generation technique in terms of space and time in our survey. We believe this comparison will help researchers and practitioners to analyze the tools and algorithms befitting to their needs. Since we did not find any survey that fits our needs, we decided to systematically identify and review existing CA generation technologies and compare them according to the above-mentioned objectives (among other things).

Table 4-I: Types of covering arrays

Covering array types	Description
Error Locating arrays (ELA) [67]	ELAs help detect faults and isolate the faulty interactions from the rest of the interactions.
Test Case Aware Covering arrays (TCACA) [63, 68]	TCACA are used when the inputs to the system are system configurations rather than actual input parameters. Once CT is applied to obtain the combination of system configurations actual test cases are then made to run those configurations. So, in order to test each configuration the entire test suite might be required, rather than a single test case as in traditional covering arrays. Having said that, some test cases might not run on specific configurations and hence the test case specific constraints might be required.
Cost aware covering arrays [63, 69]	These are the covering arrays that take into consideration the cost of testing while constructing the test suite. While generating the test suites some combinations of configurations can be more costly than others. The cost can be associated with software installations or compilations. So the cost function can be minimised while building the covering arrays.
Sequence Covering arrays [70]	These covering arrays take into consideration the sequence of occurrence of events as in GUI systems. In traditional covering arrays the order in which the values are combined to form the test suite is insignificant and will detect the same faults whereas in event driven systems the order of values is significant and hence will affect the final fault revealing power of tests.
Incremental Covering arrays [71]	Choosing the strength for a covering array can be a tricky thing to do because one cannot know a priori which strength should be chosen to find faults. For instance, a strength 2 CA can miss a fault that would be revealed by a strength 3 CA, while a strength 3 CA is typically more expensive (has more test cases) than a strength 2 CA. A solution to this trade-off problem can be incremental covering array, in which strength t is not chosen a priori but can be decided on the basis of available resources or can be incremented from a lowest strength.
l-Biased Covering array [72]	A 1-Biased CA assigns a priority to each value of a parameter (e.g., a weight), thereby giving preference to certain values so that they are covered earlier in testing. The basic idea of a l-biased covering array is that the first rows should provide the highest benefit of coverage as compared to the whole suite.

4.2 Selection protocol for conducting the orchestrated survey

We did not strictly follow established guidelines [73] to conduct a Systematic Mapping Study (SMS) [74], mostly deviating from those guidelines in the way we identified relevant publications since we did not rely on online databases such as IEEE eXplore. We nevertheless followed the SMS principles by considering research questions, establishing a precise procedure to identify relevant publications, clearly stating publication inclusion and exclusion criteria, and by defining a publication comparison framework.

4.2.1 Research Questions

Since we are interested in using a CA generation technique to produce test frames in the context of Category Partition, we identified the following research questions:

RQ1: What are the available tools/algorithms for generating combinatorial tests?

RQ2: Which techniques are used for generating covering arrays for combinatorial testing?

RQ3: Which selection criteria (to generate test frames, i.e., combinations of choices) does each tool/algorithm support?

RQ4: What is the maximum coverage strength supported by each tool/algorithm?

RQ5: Which tools support constraints and how do they represent and handle them?

RQ6: Which tools support mixed covering arrays?

4.2.2 Selection procedure

The selection procedure we followed started from survey papers [7, 17, 18, 47, 56-58], which gave a fairly good idea regarding the techniques used for generating combinational tests (Covering Arrays). However, since our objective was to search for

available tools/algorithms that support each specific technique, we first looked at the tools/algorithms mentioned in those surveys. We searched and studied literature on these tools/algorithms one by one. We extensively reviewed the related work and result sections of these papers, searching for new tools/algorithms being compared to the first list of tools/algorithms. We repeated this process multiple times, recursively, until no new tool/algorithm was identified.

Further, to ensure that our list was as complete as possible, we also searched for tools/algorithms in the papers where the survey papers were cited. We further reviewed the thesis of various researchers [72, 75-77], technical reports [78], books [52], websites (e.g., www.pairwise.org) and feature documents of various tools (e.g., ACTS, PICT).

4.2.3 Excluded papers

During the selection procedure we identified many studies which proposed an improvement over another existing algorithm, such as lowering the bound of CAs, but these papers did not have an implementation or much experimental results of comparison with other algorithms, and were not changing the essence of the algorithm to such an extent that our classification of the new algorithm would differ from that of the original. Hence, we excluded these studies. Tools/algorithms based on Orthogonal Arrays (e.g., OATS, rdExpert, reducearray2, reducearray3) were excluded because of their limitations mentioned in section 4.1. Papers on other input parameter modeling technique, e.g. classification trees (CTE_XL), Combinatorial testing for Software Product lines, Grammar based combinatorial testing, testing of compilers were also excluded. We also excluded algorithms/tools supporting prioritization of the values or parameters. We have focused on literature only in English.

4.2.4 Included papers

We have included tools and algorithms that generate combinatorial test suites. We included tools/algorithms that support input/output relationships, hamming distance-based techniques for the selection of parameters and values. We have made an exception here regarding the selection of an algorithm named Distance-Based Technique [79]. This work does not perform comparison with other tools but we have included it in our survey because it supports three selection criteria, coverage strength of 5 and is the only distance based technique we found that supports constraints. The basis of this inclusion is the variety in results.

The AETG's Web service [80] is based on the algorithm proposed by Cohen et al. [12] which was further improved by Cohen [75]. For our review, we will be considering the commercial tool AETG Web Service that is available online. ACTS [19] implements several combinatorial test generations algorithms like IPOG and IPOD [81], IPOF [82], IPOF2 [82], and IPOG-C [83] which uses constraints, all being rooted in the In parameter Order (IPO) algorithm [84]. We decided to consider ACTS itself rather than all these sub-algorithms separately.

4.3 Comparison Framework

The comparison framework consists of various comparison criteria, derived from our research questions, which we will use for comparing the tools and algorithms we selected.

4.3.1 Techniques for the generation of covering arrays

Literature shows various techniques for generating covering arrays for Combinatorial Testing. The construction of CAs are usually performed in two steps [52].

In the first step a set containing all the possible t-wise combinations is generated. In the second step the test suite is generated to cover all the combinations obtained in the first step. Both steps collectively is called the technique for test suite generation. Researchers have suggested various paradigms for the characterization of the test suite generation techniques. Grindal et al. [7] characterize techniques on the basis of the determinism of the generated output. They broadly categorized techniques as deterministic and non-deterministic and further into heuristic, artificial life based, iterative (test suite generated in iterative steps) and instant (test suite generated in one step) depending on the type of algorithm being used and how the test suite is generated. In this classification, the categories however are not disjoint. For instance, they have classified covering arrays as deterministic whereas the determinism in CA greatly depends on the algorithm used for its generation. When covering arrays are generated using Simulated Annealing the results are not necessarily deterministic [85].

Nie and Leung [47] performed an extensive survey and provided another classification scheme which classified the covering array generation techniques into greedy algorithms, heuristic search, mathematical methods and random method. It is interesting to note that, as reported by Nie and Leung, a technique may fall into more than one category: e.g., the hybrid techniques of Bryce and Colbourn [86] combines a heuristic search and a greedy algorithm to benefit from both techniques. We extended this taxonomy in our work. So in the present work we will categorize the tools and algorithms as based on: greedy algorithms, metaheuristic, adaptive random and adhoc, hybrid and algebraic techniques.

4.3.2 Test generation strategy

The algorithms for combinatorial test suite generation can be broadly categorized into Test based generation and Parameter based generation. An algorithm uses either a test case or a parameter as the building block for the generation of the test suite. In test-based generation, the algorithms proceeds by building one test at a time such that the test covers as many t-way combinations as possible thereby spanning over all the parameters. Automatic Test Case Generator (AETG) [12] falls in this strategy. Parameter based generation begins with t parameters, makes a test suite for t-wise interaction and then adds more parameters to it. While adding new parameters, new test (rows) are also added, often greedily, so that each addition leads to maximum t-way interactions in the extended set of parameters. This is the strategy of In Parameter Order [87]. The algebraic techniques, which follow a recursive approach, also use a parameter-based strategy. The building block in a recursive algebraic technique is a smaller covering array, which is a group of parameters, and the larger arrays are obtained from smaller arrays [88]. We performed a high-level analysis of the algorithms, depending on their building blocks, solely for the purpose of their classification as a test based or parameter based strategy. Specific or detailed discussion beyond this point is out of the scope of the thesis.

4.3.3 Selection criteria

In a typical situation, exercising all the possible combinations of parameter values, i.e., t-way coverage for a problem with t parameters, is simply not practical or feasible because of the large set of parameters and values. Hence, it is important to select categories (parameters), choices (values) and an appropriate selection criterion rather strategically, so that the combination of choices can lead to a manageable set of test

cases. Section 1.2.3 discusses selection criteria used mainly with the context of Category Partition namely Each Choice, Base Choice, All Combinations and Pairwise. The other selection criteria suggested in literature and surveyed by several authors [1, 7] are briefly discussed here.

The Uniform Strength Interaction criterion, or t-wise, criterion requires that any combination of values belonging to t parameters should be combined at least once in the test suite. Here all the parameters are supposed to be uniformly integrated with a constant value t [89]. The Pairwise criterion mentioned in the context of Category Partition Technique corresponds to a uniform strength of 2.

The Variable Strength Interaction criterion, or Mixed Strength Interaction, is an extension to the t-wise criterion that requires t-wise interaction among a subset of parameters and q-wise interaction among the remaining parameters [90].

Instead of exercising interactions of the complete set of parameters, an Input Output Based Interaction criterion splits the set of parameters into (possibly overlapping) subsets such that each contain the parameters that impact the value of one output parameter [50]. Although the authors used All Combinations criterion but any selection criteria can be used on each subset of parameters and results can be combined to obtain complete test cases.

The goal of the Distance Based criterion is to select combinations of parameter values, i.e., test cases, that are as diverse as possible, diversity being measured as the distance between those test cases, for instance using the Hamming distance [3]. The procedure starts by creating the binary representation of the input space and then

selecting inputs that are maximum Hamming distance apart from each other and include them in a test frame. A greater hamming distance symbolizes diverse test cases.

The Random Input criterion selects a randomly chosen number of test cases and each test case is a random selection of parameter values.

4.3.4 Coverage Strength support

The strength of a CA is the number of parameters whose interactions are exercised by a specific CA. By extension, the maximum strength of a CA generation algorithm is the maximum strength value of CAs generated with this algorithm as reported in experiments in the literature. Studying coverage strength support is important since increasing strength has been experimentally shown to relate to fault detection [52, 78]. In order to obtain such a maximum strength we relied on the already published literature, either the paper where the technique/algorithm was first presented or subsequent work where that algorithm was used for comparison. We took the maximum of all the values we collected as published in the respective literature along with their test configuration (i.e., number of parameters, number of values of those parameters, …) that was used to obtain these values. This is explained further in the result section 4.4.4.

4.3.5 Constraint support

Constraints are limiting the construction of a CA by forbidding some combinations. One can distinguish between environment constraints and system constraints [52]. An environment constraint is an intrinsic characteristic of the application domain, i.e., the parameters and their values, and is strictly forbidden, e.g., Linux OS can never be combined with Internet Explorer browser. A system constraint on the other hand is user defined (e.g., one user cannot select a value less than 10). Such a constraint is specified,

for instance, to reduce the number of test cases and therefore cost, or relaxed, for instance to test the robustness of the system.

Constraints can be represented either as forbidden tuples, allowed tuples or formally specified with the help of propositional formulas or logical expressions using Boolean, relational or arithmetic operators [83, 91]. A forbidden tuple is a combination of parameter-values that cannot appear in the final test suite. A single constraint can give rise to any number of forbidden tuples [15]. In a typical situation, constraints are formally specified, not necessarily explicitly, by the test engineer. Tools/algorithms which accept formally specified constraints are therefore more usable than those that do not since in the latter case, the test engineer needs to remodel the constraint input to transform the formal specification into a list of forbidden or allowed tuples [15].

As an example, we demonstrate the constraint representation by three combinatorial testing tools; CASA [19, 53], ACTS [19] and PICT [21]. We take a constraint example from our industrial Level 2 case study where the constraint is specified on choice p21 of category 2 (Appendix C.11): p21→ (p12||p13) && p41 && (p37||p38||p39). The constraint A → B means that constraint B applies on choice A. Therefore, the constraint on choice p21 is a combination of choices from categories 1 (e.g., p12), 3 (e.g., p37) and 4 (i.e., p41).

The representation of this constraint in CASA is similar to logical implication A→ B, given that such a constraint can be written as A→B = !A || B. In order to write the constraint in CASA, it is converted into a Conjunctive Normal Form as follows:

!p21 || ((p12||p13) && p41 && (p37 ||p38||p39))

= !p21||p12||p13 && !p21 || p41 && !p21 || p37||p38 ||p39

This will hence become three constraints for CASA in CNF.

The constraint on p21 is represented in ACTS as follows:

Category2= “p21”=>((category1= “p12” || category1= “p13”) && (category 4 = “p41”) && (category3= “p37” || category3=“p38” || category3 = “p39))

In PICT the constraint is represented as follows:

IF ([Category2]="p21") THEN ([Category1]="p12" OR [Category1]="p13") AND ([Category4]="p41") AND ([Category3]="p37" OR [Category3]="p38" OR [Category3]="p39");

These three examples show how the constraints can be specified using different tools. We will use these representation techniques to specify constraints among choices for our example academic and industrial case studies (section 5.3)

We recognized four mechanisms for handling constraints by the tools/algorithms. The first mechanism is handling constraints before executing a specific test generation algorithm. This mechanism can be adopted for those algorithms that accept ‘allowed tuples’. Allowed tuples are the set of values that are permitted to appear together. Only the allowed tuples are given as an input to the test generation algorithm and the algorithm does not need to be modified to check infeasibility. The second mechanism is to replace the invalid test cases with valid ones once the test suite has been generated using a specific technique [55]. The third mechanism is to integrate constraint handling into the CA generation algorithm with an ad-hoc procedure. The last mechanism is to integrate a SAT solver to the algorithm generating a combinatorial test suite in order to select valid tuples as per the constraints.

4.3.6 Support for mixed covering arrays

A typical software system will have a large number of parameters and each parameter will not necessarily have the same number of values. Therefore, a tool/algorithm should be able to support mixed covering arrays to cater to the needs of such a software system.

4.4 Results

In this section, we will answer the research questions, individually. The results are further analyzed in section 4.5. Appendix A.1 and A.2 contains the complete list of 75 algorithms/tools obtained as a result of the selection protocol.

4.4.1 RQ1: What are the available tools/algorithms for generating combinatorial tests?

Figure 4-1 shows the number of tools/algorithms we found over four years intervals from 1991 (the earliest year we found) to 2014. The total number of tools/algorithms obtained using our search protocol is 75 (Appendix A.1 and A.2). Among the first tools for generating tests is T-Gen [92], introduced in 1991, and based on the Category Partition technique [4]. In the next four years no tool was proposed and then from 1999 onwards there has been a constant rise in the number of tools/algorithms for generating covering arrays for combinatorial suiting. This clearly marks the importance of functional testing and a need to have an optimal test suite. 69% of the tools/algorithms have been proposed in the last eight years.

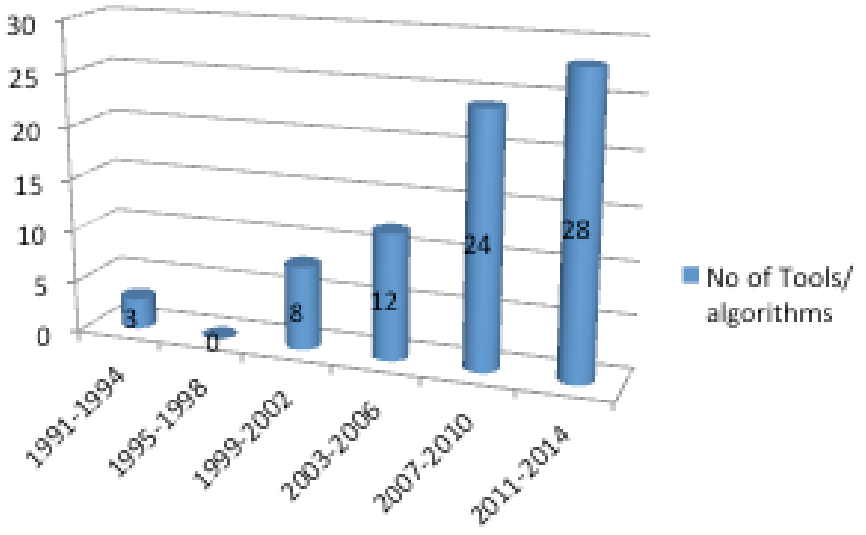


Figure 4-1: Number of tools/algorithms identified over years, presented over four year

While searching for specific tools/algorithms we observed that authors suggested improvements to their own technique over the years while proving experimentally that their new algorithm was doing better than before. In our study, an algorithm with multiple references corresponds to the improvements the algorithm went through, along the way, and we considered the results of the latest upgrade.

4.4.2 RQ2: What covering array generation techniques are used by the tools and algorithms discovered in RQ1?

In this section, we discuss various techniques used for the generation of covering arrays and used by tools and algorithms discovered in section 4.4.1. Section 4.4.2.1 discusses various techniques followed by the tools and algorithms that uses the specific technique. We then look at the strategy used by each tool or algorithm i.e., test-based versus parameter-based strategy (section 4.4.2.2). Lastly, we study the various types of meta-heuristic techniques (section 4.4.2.3) used by tools or algorithms.

4.4.2.1 Covering array generation techniques

We identified five different types of techniques used for the generation of combinatorial test suites: Greedy Techniques, Meta-Heuristic Techniques, Adaptive random / Adhoc techniques, Hybrid Techniques and Algebraic techniques.

A Greedy Technique generates tests by constructing a locally optimal solution and ensures that each new test uses the maximum possible uncovered combinations. They are usually faster than the Meta Heuristic techniques but do not always produce the smallest test suites. These algorithms return a local optimum rather than a global optimum. We assign tools/algorithms based on backtracking algorithms, branch and bound techniques, exhaustive search, AETG type algorithm [12], IPO based algorithms [87] etc. to this category.

The generation of a combinatorial test suite is an optimization problem and meta-heuristic techniques are known to be good at solving such problems. These techniques can be evolutionary algorithms, e.g., Genetic Algorithms, or naturally inspired algorithms, e.g., Particle Swarm Optimization, or any other standard known optimization algorithms. Such an algorithm searches the neighborhood of a solution and finds the best fit. The algorithm starts from a preexisting test called a seed and, after performing a series of transformations to achieve an optimum test suite. A heuristic search such as Simulated Annealing produces smaller test sets than a greedy algorithm but takes more time to execute [53]. The various meta-heuristic techniques we identified and are used for generating covering arrays are Hill Climbing, Simulated Annealing, Tabu Search, Genetic Algorithm, Ant Colony Optimization, Partial Swarm Algorithm, Harmony Search, Extremal Optimization and Great Flood.

The category of Adaptive random or ad-hoc techniques contains two types of techniques. Adaptive random uses an algorithm that relies on a measure of distance between the parameter values, e.g., using the Hamming distance, to generate the test cases that are maximally apart from one another (e.g., [79, 93]). The set of ad-hoc techniques contains those tools and algorithms that are not using any of the other techniques. An ad-hoc approach typically selects the test cases randomly or on the basis of some input distribution (e.g., [94, 95]).

Hybrid approaches are proposed by researchers to achieve better and optimal results by combining the benefits of two different types of techniques. The objective behind combining techniques is to reduce the size and generation time of the covering array and increase the coverage and hence the fault detection. For instance, Bryce et al. [86] combine a greedy algorithm with a heuristic search, Cohen et al. [96] combine a mathematical approach with Simulated Annealing.

Algebraic Techniques create covering arrays by either directly computing a mathematical function or by using defined rules [97]. Some algebraic constructions use recursion to obtain larger covering arrays from smaller building blocks [88]. The smaller building blocks are subset of parameters that are grouped together to form larger covering arrays. Algebraic techniques for creating covering arrays can also be extensions to mathematical methods for constructing orthogonal arrays [47, 52, 98]. However, the applicability of algebraic approach is limited because they impose restrictions on the system configurations that they can accept.

Figure 4-2 shows that, out of these 75 algorithms, 40 (53%) use a greedy approach for the generation of the combinatorial test suite, 13 (17%) use Meta heuristic techniques,

five (6%) belong to the category of Adaptive random and adhoc. Six (8%) tools/algorithms use a hybrid approach, either combining a greedy technique and a metaheuristic technique, or a greedy technique and an algebraic technique. Only four (5%) tools/algorithms support algebraic techniques, probably because algebraic techniques are not as versatile as other techniques (e.g., their support for different strength values or their support for constraints). We also found seven (9%) tools which were not accompanied by a detailed technical documentation which could help us classify them according to this comparison criterion.

The advantage of greedy and meta-heuristic technique is their application to any size of system configurations, i.e., there is no restriction on the number of parameters or the number of values each parameter can take. The downside is that they take more time to create a covering array [52] as compared to the algebraic techniques. On the other hand, algebraic techniques are extremely fast and lightweight but only on a subset of system configurations. They cannot, as well, deal efficiently with constraints [47].

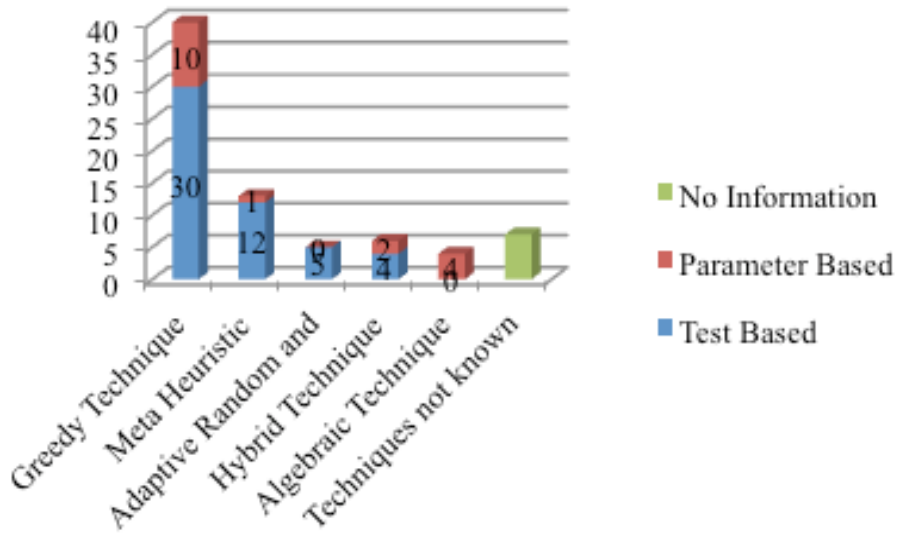


Figure 4-2: Categorization of the tools/algorithms based on techniques and

4.4.2.2 Test based vs parameter based strategy

We observed that out of the 68 tools/algorithms (75 minus seven for which we do not have technical information), 75% of the tools (51 out of 68) followed a test-based generation strategy and 25% (17 out of 68) followed a parameter-based generation strategy. Out of the 40 tools/algorithms that generated test suites using a greedy approach, 30 followed a test-based generation and 10 followed a parameter-based generation. Figure 4-2 summarizes those results.

4.4.2.3 Meta-heuristic techniques

Figure 4-3 shows the number of tools using a specific metaheuristic technique for the generation of covering arrays. The tools/algorithms which use meta-heuristic techniques either belong to the category of meta-heuristic or to the category of hybrid (Figure 4-2). Three meta-heuristic techniques namely Particle Swarm, Genetic algorithm and Simulated Annealing are more widely used than others.

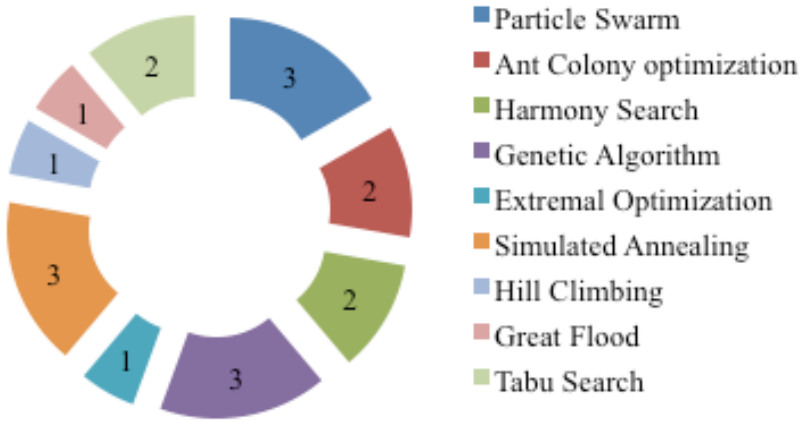


Figure 4-3: Number of tools/algorithms using different metaheuristic algorithms

4.4.3 RQ3: Which selection criteria (to generate test frames, i.e., combinations of choices) does each tool/algorithm support?

We identified seven different selection criteria supported by the 75 tools/algorithms: Base Choice, Each Choice, Input/Output, Distance, Uniform Strength, Variable Strength and Random.

The Base Choice criterion requires the identification of a base choice for each category, which is a choice that is considered the most important of the choices of a category. Since identifying the base choice of a category can also be implicitly done by assigning weights to the category's choices and selecting the best (max or min, depending) weighted choice as base choice, we classified all the tools/algorithms which support the assignment of weights to parameter values in the Base Choice criterion category.

We also made a difference between the Each Choice criterion and the Uniform Strength criterion. Uniform Strength typically means a strength t of at least two (t>=2) where as Each Choice corresponds to uniform strength of strength one (t=1). A tool supporting uniform strength of at least two would support uniform strength of one. In our analysis we put the tools/algorithms which explicitly mention their support for the Each Choice criterion (uniform strength one) in a separate Each Choice criterion category. During our research we also found a few tools that support variable strength CAs without specifically mentioning support for uniform strength. However since the former implies the latter, we classified those tools as variable strength and graphically included the variable strength category into the uniform strength category (Figure 4-4).

Obtained results are shown in Appendix A.3 and are summarized graphically in Figure 4-4: 72 (96%) of the tools support uniform strength, 24 (32%) support variable strength (and therefore uniform strength). Nine tools support three criteria: Input/Output, Variable Strength and Uniform Strength. Another three tools, ACTS [19], PICT [21] and IBM Focus [91], support Uniform Strength, Variable Strength and Base Choice. Out of these tools, IBM focus and PICT support assigning weights to values. Tcases [99] supports four criteria, which is the maximum we found: Uniform Strength, Variable Strength, Random and Each Choice. Two tools support Distance based: [93] and [79]; the latter also supports Random and Uniform Strength whereas the former supports Uniform Strength.

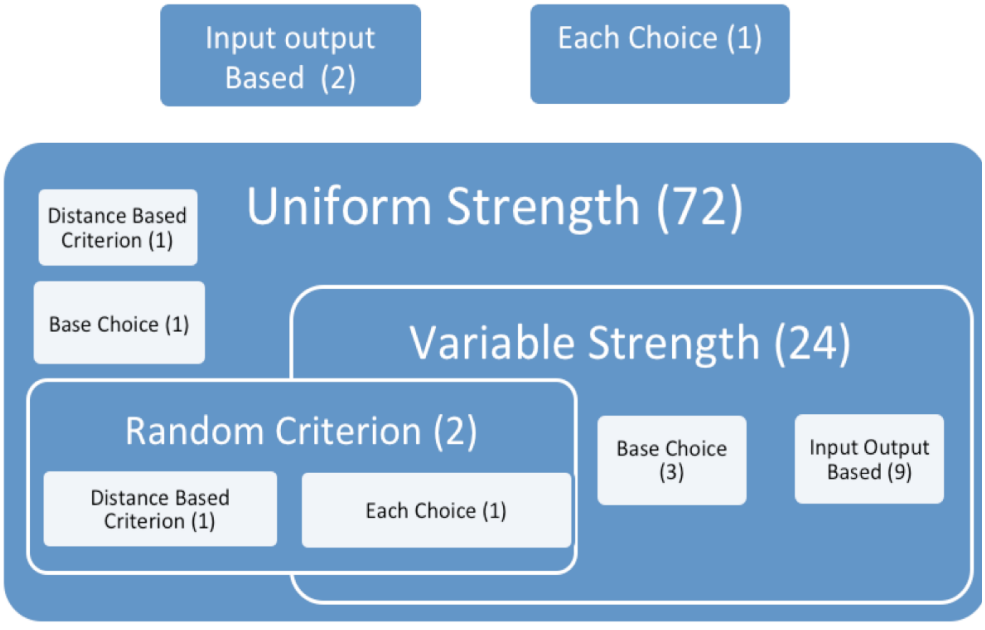


Figure 4-4: Support of the tools for the selection criteria

4.4.4 RQ4: What is the maximum coverage strength supported by each tool/algorithm?

We studied maximum strength collectively for Uniform Strength and Variable Strength tools/algorithms (Appendix A.4). For a tool/algorithm which only supports Uniform Strength, the highest strength is obtained for a specific test configuration. In case results were available for more than one test configuration, we chose that configuration, from variable strength or uniform strength, which supports the highest strength value and mixed covering array with maximum number of parameter values.

Figure 4-5 shows the result for 72 tools: three tools do not support Uniform Strength (Figure 4-4) and so are excluded for this question. Out of 72 tools, 26 (36%) support a maximum strength of two, 14 (19.5%) the strength of three, 11 (15%) the strength of six, three (4%) the strength of 12 and we found one tool (Harmony Search Strategy [65]) which supports the strength of 14.

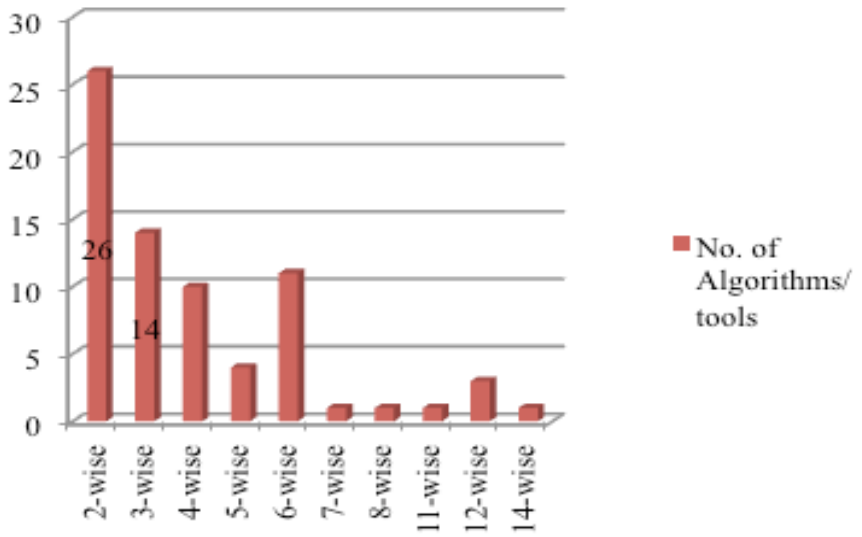


Figure 4-5: Maximum coverage strength supported by each tool/algorithm

For the selection of research papers to answer this question we followed the following approach. We obtained the strength from two types of sources; the strength experimented in researcher's own work and/or any other research work in which a comparison is made with that specific tool. We have taken the higher strength of the two and included it in our analysis. There were certain tools/algorithms for which the results were not shown or detailed information was not available but the authors claimed that their algorithm supported a certain strength. We used values reported by authors but flagged the papers in Appendix A.4. Further, ATD [100] did not have experimental results but the authors claim to support t-wise coverage, for this tool we assumed the most common value of t=2.

Even though AETG Web Service [80] is based on Cohen et al. technique [12], which supports Uniform Strength, random inputs and All Combinations as the selection criteria, the AETG Web Service only explicitly supports the first criterion of those three. Based on our paper selection criteria (section 4.2), we only consider the AETG Web Service and only consider its support for Uniform Strength.

4.4.5 RQ5: Which tools support constraints and how do they represent and handle them?

We found 32 (44%) tools/algorithms supporting constraints (Appendix A.5). The two important aspects of constraint support we are focusing on are representation and handling mechanism.

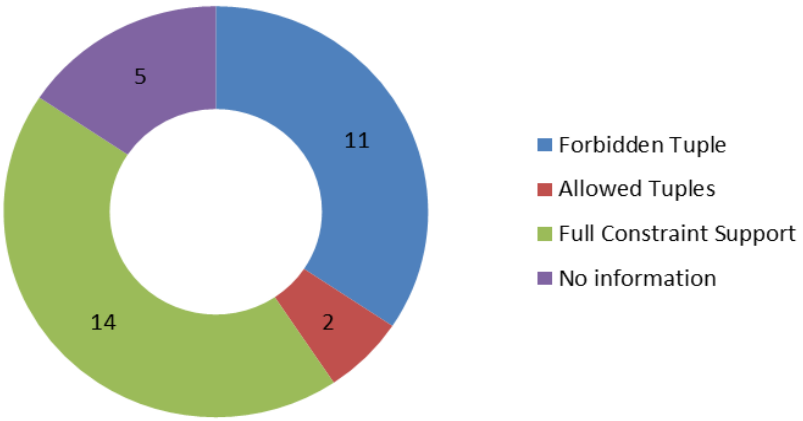


Figure 4-6: Number of tools/algorithms supporting a specific constraint representation

Tools supporting constraints require an input under the form of forbidden tuples (11, i.e., 34.3%), allowed tuples (2 i.e., 6%) or full constraint support (14, i.e., 43.7%): Figure 4-6. For five tools, available documentation indicates support for constraints but fails to provide further details so we can classify.

Figure 4-7 summarizes the mechanisms to handle constraints during CA construction. 59% (13+6) of the tools have a mechanism embedded in the CA

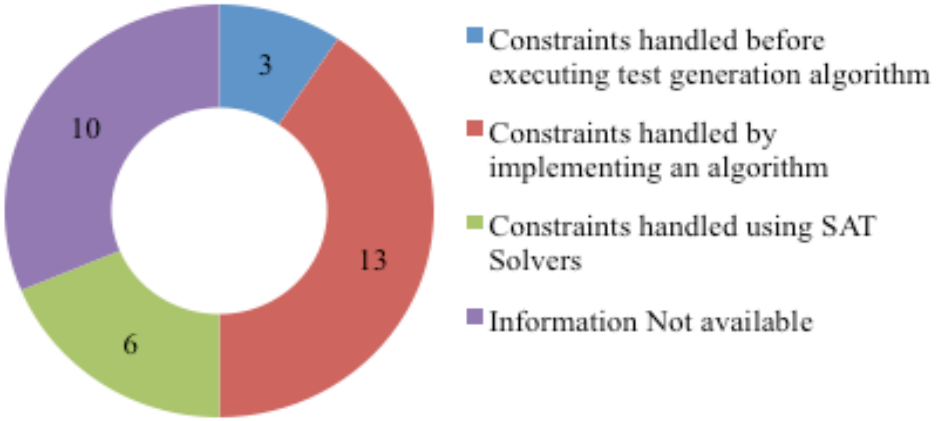


Figure 4-7: Number of tools/algorithms supporting a specific constraint handling mechanism

construction algorithm to handle constraints: 40% use an ad-hoc algorithm, 19% use a SAT solver. None of the tools has been found to use the mechanism of replacing invalid test cases once the test suite has been generated. Certain tools did not have enough information in the technical documents, in order to help us make decisions.

In our study we found three tools which supported robustness testing, i.e., test for the invalid values, without requiring that specific out-of-bound choices be specified as input: PICT [21], PictMaster [101] and IBM Focus [91].

4.4.6 RQ6: Which tools support Mixed Covering Arrays?

Figure 4-8 shows the tools/algorithms which support mixed covering arrays as compared to the tools/algorithms which support only traditional covering arrays: see detail in Appendix A.6. 76% of the tools support mixed covering arrays, i.e., parameters can have varying numbers of values, whereas 5% of the tools/algorithms assume all parameters have the same number of values. We were not able to collect information for 14 tools.

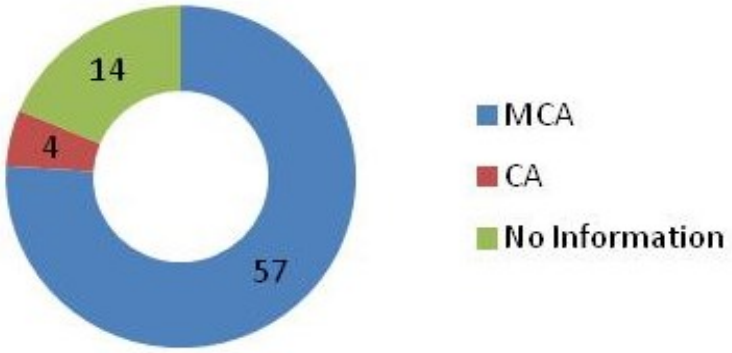


Figure 4-8: Support for mixed covering arrays

4.5 Discussion

In this section we combine research questions to get a better picture of algorithms/tools' capabilities.

4.5.1 Combining RQ1, RQ2 and RQ3

The objective of this combination is to analyze the technique and the selection criteria that technique mainly supports: Table 4-II. We observe that two of the criteria, namely Base Choice and Input/Ouput, are only supported by greedy algorithms and by none other technique. 66% of the algorithms (16 out of 24) which support Variable Strength are using a greedy technique. On the other hand, only 25% (6 of 24) of the tools which support Variable Strength use a heuristic technique. A similar trend is observed for Uniform Strength; out of the total 72, 38 algorithms (52%) use a greedy technique, 13 algorithms (18%) use a heuristic technique, six (8%) use a hybrid technique whereas four (5%) and five (7%) algorithms respectively use an algebraic or an adaptive random technique. It is important to note here that algebraic techniques contribute to the

Table 4-II: Number of tools/algorithms on the basis of techniques and selection criteria

	Greedy (Parameter)	Greedy (Test)	Meta-Heuristic (Parameter)	Meta-Heuristic (Test)	Algebraic (Parameter)	Adaptive Random and adhoc (Test)	Hybrid (Parameter)	Hybrid (Test)	Don't Know
Each Choice									2
Base Choice	1	3							
Variable Strength	3	13		6					2
Uniform Strength	10	28	1	12	4	5	2	4	6
I/O Based criteria	1	10							
Distance Based criteria						2			
Random criteria						1			1

generation of covering arrays using only the Uniform Selection criterion and do not support any other criterion. Similarly, hybrid techniques only support Uniform Strength. Only adaptive random and adhoc techniques provide support for Distance based and Random criteria. None of the tool supports All Combinations.

We conclude that greedy techniques largely support the generation of covering arrays using multiple selection criteria, i.e., Base Choice, Variable Strength, Uniform Strength and Inout/Outout based, whereas heuristic techniques support only Variable Strength and Uniform Strength.

4.5.2 Combining RQ1, RQ2, RQ4

The objective of this combination is to identify the coverage strength support of each technique: Table 4-III. We observe that greedy techniques support a range of strengths varying from 2 to 12. The higher strengths in greedy techniques are supported by test-based generations: GTWay [102], GVS [103] and ITTDG [104]. The test configuration for these algorithms used 12 parameters, with a maximum number of values of 10 for two parameters. It is important to mention here that two of these algorithms, i.e., GVS and ITTDG, support three selection criteria (Variable Strength, Uniform Strength and Input/Output), which clearly shows that greedy techniques have outperformed other techniques on the basis of support of selection criteria and higher strength values. The highest strength in our survey was however supported by an algorithm named Harmony Search Strategy (HSS) [65] with a strength of 14. HSS uses a meta-heuristic technique and test-based generation for generating covering arrays. The HSS algorithm supports Variable Strength and Uniform Strength and obtains a strength of 14 with a subset of parameters while using the variable strength selection criterion.

Table 4-III: Number of tools/algorithms on the basis of techniques and coverage strength

Selection criteria	Greedy (Parameter)	Greedy(Test)	Meta-Heuristic (Parameter)	Meta-Heuristic (Test)	Algebraic (Parameter)	Adaptive and Random adhoc (Test)	Hybrid (Parameter)	Hybrid(Test)	Don't Know
2-wise	2	10	1	4	2	3			4
3-wise	1	7		4			1		1
4-wise	1	2			2	1		4	
5-wise	3					1			
6-wise	2	5		2			1		1
7-wise				1					
8-wise		1							
11-wise	1								
12-wise		3							
14-wise				1					

We also observe from Table 4-III that algebraic techniques support a maximum strength of four whereas IPOD [81], which is a hybrid of an algebraic technique and a greedy parameter-based technique, supports a strength of 6. Similarly, the hybrid of a meta-heuristic technique with a greedy technique has also elevated the strength support of meta-heuristic techniques to 4 with an exception of Tabu search. Selecting one technique over another should also consider other factors such as the size of the CA generated or the time it takes to generate it. This is however beyond the scope of the present work.

4.5.3 Combining RQ1, RQ2, RQ5

The objective of this combination is to know which technique supports constraints: Figure 4-9. 53% of the algorithms/tools (17 of 32) which support constraints use a greedy technique. These algorithms/tools either implement the constraint handling algorithm or use a SAT solver for handling the constraints. This is followed by 13% of the algorithms that use meta heuristic techniques (e.g., simulated annealing). A meager number of tools

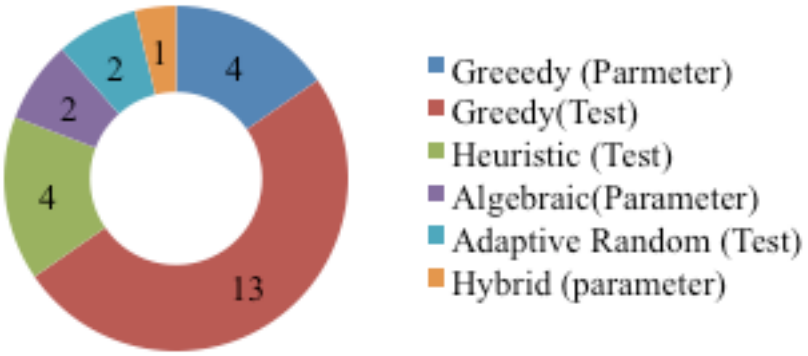


Figure 4-9: No. of tools/algorithms based on techniques and constraint

based on algebraic (2 tools), adaptive random (2 tools) and hybrid techniques (1 tool) support constraints. Hence, we conclude from this observation that algorithms based on greedy technique being in majority outnumber other techniques in providing support for constraints, which is followed, by metaheuristic techniques.

4.6 Conclusion

Functional testing from a plain English specification, for instance following Category Partition, requires that one identifies parameters, categories, choices and then combine those choices according to some selection criteria, while accounting for constraints on choices, to generate test frames and eventually generate test cases. Covering arrays have been used for a long time to generate such combinations. Covering arrays come in various forms and have various capabilities and it is difficult to identify which covering array generation technique is the most suitable to the problem of generating test frames for Category Partition technique. When faced with this problem, we searched for a solution and did not find enough data to make an enlightened one. We therefore decided to perform a systematic survey of technologies supporting covering array generation. We report in this chapter on the procedure we followed in this

systematic survey and on the procedure we followed to characterize the covering array technologies we have found.

We eventually identified 75 covering array generation tools and algorithms. Our comparison framework allowed us to make a number of observations.

We observe that different covering array construction technologies support different sets and numbers of selection criteria in different amounts: 43% of the greedy techniques support up to three criteria; 46% of the meta-heuristic techniques support two criteria; 40% of the algorithms based on adaptive random techniques support up to three criteria. We believe these differences are not intrinsic to the construction technologies; for instance, there is no reason to believe that meta-heuristic techniques (or hybrid ones) could not support the complete list of criteria we have listed previously in the chapter, or higher strength values (at the expense perhaps of longer execution times); we conjecture greedy algorithms have been so far popular due to their simplicity. Some technologies support very high strength values (up to 14), and 70% of the tools do not support a strength greater than four. The cost-benefit of such values is, as far as we know, yet to be confirmed experimentally. We found that only 44% of the 75 tools support constraints, and that constraints are provided mostly either as forbidden tuples or formal specifications. Constraints are mostly handled by greedy construction techniques; however, again, there is no reason to believe other techniques could not equally handle constraints.

We observe that although metaheuristic, adaptive random/adhoc and algebraic techniques form a smaller part of the tool supporting covering array construction, they are equally focused on advance features for creating covering arrays as greedy techniques.

On the other hand, tools/algorithms based on greedy techniques are plenty in number, which can be attributed to the fact that they are flexible to implement. They support large system configurations including constraints, selection criteria, mixed covering arrays and higher strengths, which is essentially a requirement for software testing.

Going back to our problem on identifying CA construction technique to support Category Partition, whereby one needs that technique to handle constraints, variable numbers of choices per category (i.e., values of parameters), and selection criteria including at least Pairwise, we can conclude that a greedy algorithm is likely the best choice to date as this kind of technique supports selection criteria, various strength and constraints. In case there are few or simple constraints, a user may be able to spell out forbidden tuples and use a greedy algorithm that accepts such input (e.g., [105-107]); in case of complex or numerous constraints, manually constructing forbidden tuples may not be practical so a greedy algorithm that uses an adhoc algorithm for constraints (e.g., [21, 91]) or that incorporates a SAT solver (e.g., [19, 37, 108]) may be the ideal choice.

Chapter 5 —Category Partition Design of Academic and Industrial Case Studies

We selected a number of case studies to observe the behavior and quality of test suite generated using different selection criteria and different technologies for the generation of test frames. These case studies are broadly categorized as academic and industrial. The academic case studies are well known to the testing community. We obtained the industrial case studies from our industry partner, Ericsson Inc. The basic intent of using different case studies is to have a variety in terms of size, number of constraints, complexity, CP specifications and most importantly their use as a toy project or a live project.

5.1 Description of Academic case studies

We used various academic case studies, which are easily available in many software testing text books [109, 110]. In the next section, we discuss the Next Date problem (section 5.1.1), the Triangle problem (section 5.1.2) followed by the discussion on the PackHexChar problem (section 5.1.3).

5.1.1 The Next Date Problem

The next date case study finds the next date given an input date. There are three variables in this problem: year, month and day. The Georgian Calendar starts from 1582, therefore we chose this year as a starting year, and we make an assumption for the end date to be 2100. Therefore, the range of valid values would be [1582, 2100]. Any date falling outside this range would be an error.

January, March, May, July, August, October and December have 31 days, while April, June, September and November have 30 days. The month February has 28 days except on leap years when it has 29 days. A leap year is calculated based on divisibility by 4 and 100. If a year is divisible by 4 and not divisible by 100, then it is a leap year. If the year is divisible by 4 and divisible by 100 then this year is a leap year only if the year is divisible by 400 as well. So for example, the year 1986 is not divisible by 4 and not divisible by 100 so it is not a leap year, and the year 1996 is divisible by 4 and not divisible by 100, therefore this is a leap year. The year 1900 is divisible by 4 and is divisible by 100, but it is not divisible by 400, therefore it is not a leap year, however the year 2000 is divisible by 4 and divisible by 100, and by 400, therefore it is a leap year.

5.1.2 The Triangle case study

The triangle problem is a classic academic problem to determine if three integers SA, SB, SC each corresponding to the three sides of a triangle, represent a valid triangle or not. If it is a valid triangle, the program further determines the type of the triangle, i.e., scalene, isosceles or equilateral. Therefore, the output of the program would be the type of the triangle, i.e., equilateral, isosceles, scalene or not a triangle. In order to determine if the sides of the triangle are valid and can form a triangle, the triangle inequalities play a role. Three sides SA, SB, and SC denote a triangle if and only if the triangle inequalities hold, that is: SA < SB+SC; SB < SA+SC; SC < SA+SB.

The inputs of the system have to be greater than zero in order to determine the type of the triangle. The following is how we determine the type of triangle:

1. If all three inputs are equal, then the triangle inequalities hold and the triangle is equilateral.

2. If exactly two sides are equal, and the triangle inequalities hold, then the triangle is isosceles.

3. If no pair of sides are equal, and the triangle inequalities hold, then the triangle is scalene.

4. If none of the above conditions is met, i.e., the triangle inequalities do not hold, the output is notATriangle.

5.1.3 The PackHexChar case study

PackHexChar is a Java implementation of the sreadhex routine from the GhostScript program which processes the PostScript page description language [110]. This Java adaptation of the sreadhex routine, takes in a string of characters representing hexadecimal digits and packs two hexadecimal characters into its equivalent binary representation. For example, string of hexadecimal characters "56AB" that has 4-bit binary representation of 0101, 0110, 1010 and 1011 is packed into two 8-bit binary representation of 01010110 (binary representation of 56) and 10101011 (binary representation of AB). The PackHexChar implementation then returns the decimal equivalent of the two 8-bit binary representations that is 86 and 171. If the input string contains characters other than the hexadecimal characters, they are ignored. Thus, "56mAB" will yield the same results as 56AB.

In addition to the decimal equivalent of the array of bytes, the implementation also returns an integer value depending on the number of characters in the string. If the input string contains an even number of hexadecimal characters, the program pairs the hexadecimal characters and returns -1 as the integer value. If the input string contains an odd number of hexadecimal characters, the program after pairing even hexadecimal

characters, returns the remaining hexadecimal character. For instance, for the input string “abcde’, the program groups “ab” and “cd” and returns the decimal equivalent of “e”.

The program also takes in as input a variable RLEN. RLEN lets a user analyze only a substring of the input string and packs the hexadecimal characters in that substring. The program returns an integer value of -2 for illegal value of RLEN, i.e., if the value of RLEN is greater than the string length or is negative. If there is no hexadecimal character in the RLEN substring the program returns an integer value of -4.

The program also lets a user append a hexadecimal value in the beginning of the string. This is particularly useful when the string is split and analyzed in pieces with repeated calls to the PackHexChar program. If the number of characters in RLEN substring is odd, this split can result in a carry forward hexadecimal character. This hexadecimal character is appended to the beginning of the string for the next call to the PachHexChar program. The program uses input variable ODD_DIGIT to implement this feature. An ODD_DIGIT value of -1 indicates that no character is to be appended whereas an ODD_DIGIT value of 0 through 16 refers to the corresponding hexadecimal character which is appended to the beginning of the string. Any illegal value of ODD_DIGIT (negative or non-hexadecimal equivalent) is the basis for the PackHexChar program to return an integer value of -3.

5.2 Description of Industrial Case Studies

We chose industrial case studies to see the impact of implementation of the Category Partition technique on real data. These case studies, designed at Ericsson Inc., are a part of a middleware in the telecom domain and relate to the radio protocol architecture of LTE.

The CP specifications and the constraints are derived from technical documents, LTE standards, algorithmic specifications and internal documents. Further, to clarify the implementation, we also looked at the product code, original test suite and finalized the CP specification after regular meetings with experts at Ericsson. We implemented the CP technique at the same level of granularity as the original test suite. This helped us in analyzing the limitations of the test suite designed by Ericsson engineers and suggesting a CP specification for the functionalities, thereby facilitating comparison between the two test suites. All the CP specifications collectively at the end will result in the testing of the entire product, akin to the original test suite. For confidentiality reasons we cannot disclose the actual name of the functions and their detailed functionality neither we can disclose the detailed CP specifications. We have replaced the name of the files, parameters and categories for the same reason. We will however discuss the size of CP specifications in terms of number of parameters, categories and choices, the size of constraints implemented on the case studies, i.e., the number of choices involved in the constraints and the number of tuples involved in the constraints, along with the code size of the implementation of the functionality. Once the parameters, categories and choices are specified, constraints are defined using the representation discussed in section 4.3.5.

Since these software units collaborates with many other units and, for performance reasons, are not designed by means of defense programming but through contracts, they therefore assume their input parameters do not go out of bounds and therefore we did not specify any error choices in the CP specifications.

We decided to implement Category Partition at two levels of functionality implementation: Level 1 and Level 2.

5.2.1 Level 1 case studies

At the lower level, renamed in the rest of the discussion as Level 1 (or L1), we use CP on the functionalities provided by library functions. These functions implement algorithms further required by functionalities that are more complex.

The Level 1 case studies are based on calculating the possible Orthogonal Frequency Division Multiplexing symbols (OFDM) for Physical Downlink Control CHannel (PDCCH). Each Level 1 case study is a combination of a number of functionalities in the form of product files. There are eight product files in this set as shown in Table 5-I, which corresponds to 10 cases studies (CS_L1_A to CS_L1_J). These product files are being tested either as one unit (consisting of multiple functions) or by testing one function at a time, depending on the functionality they implement. As mentioned earlier our level of granularity is inspired by the original test suite, which made it easy to compare the original test suite with the CP generated test suite.

5.2.2 Level 2 cases study

The Level 2 case study is a bigger functionality which calls a number of level 1 functions. At the higher level, renamed as Level 2 (or L2), we use CP on functionalities that rely on functions in Level 1 in addition to other functionalities. There is one product file for this Level, i.e., Product File 9, which corresponds to case study L2_A (Table 5-I).

We discuss the results for Level 1 and Level 2 separately. Table 5-I specifies the nine product files and their corresponding CP specifications names for both levels. These CP specifications are detailed in Appendix C. Depending on the functionality and the original test suite (for the purpose of comparison), a product file can have two CP

Table 5-I: List of case studies derived from the product files of Level 1 and Level 2

Case study	Product Files
Case Study L1_A	Product File 3
Case Study L1_B	Product File 3
Case Study L1_C	Product File 2
Case Study L1_D	Product file 6
Case Study L1_E	Product file 7, Product file 8
Case Study L1_F	Product file 1
Case Study L1_G	Product file 7
Case Study L1_H	Product file 4, product file 7
Case Study L1_I	Product file 4
Case Study L1_J	Product file 5
Case Study L2_A	Product file 9

specifications and therefore can form two different case studies or two product files can form one CP specification (e.g., case study L1_E) as shown in Table 5-I.

In Chapter 6, we discuss the results of applying the Category Partition Technique on all the industrial case studies. We use the Next Date and Triangle case studies along with the industrial case studies CS_L1_A, CS_L1_C and CS_L2_A in Chapter 7 for observing the impact of different types of constraints on the quality of test suites while using Category Partition testing. In Chapter 9, we use PackHexChar along with Next Date, Triangle and CS_L1_A, CS_L1_C and CS_L2_A for comparing the quality of the proposed criteria (Constrained Base Choice and Extended Constrained Base Choice) with existing criteria (Base Choice, Each Choice, Pairwise and Three-way).

5.3 Category Partition specifications of the case studies

In this section, we discuss the CP specification of both academic and industrial case studies.

Table 5-II: Characteristics of CP Specifications for academic case studies

							Total Choices involved in X constraints														
Case study	LOC	Parameters	Categories	Choices	Single Choices	Error Choices	X= All Constraints	X= 1 constraint	X= 2 constraints	X= 3 constraints	X= 4 constraints	X= 5 constraints	X= 6 constraints	X= 7 constraints	No. of terms in CNF	2-tuple constraints	3-tuple constraints	4- tuple constraints	6- tuple constraints	% of choices constrained	Constraint per choice
Triangle	38	3	9	18	3	3	9	3	0	0	6	0	0	0	12	9	3	0	0	66	1.5
NextDate	179	3	4	18	1	6	6	4	1	1	0	0	0	0	4	3	1	0	0	72	0.84
PackHexChar	211	3	7	32	0	4	26	13	5	1	2	3	1	1	26	22	0	3	1	94	1.0

5.3.1 Characteristics of CP specifications of Academic case studies

We follow the process of deriving a Category Partition specification for each case study, involving the identification of parameters, categories, choices and constraints among the choices, and including single and error annotations. The constraints among the choices are represented as discussed in section 4.3.5.

Appendix B contains the complete CP specification of the three academic case studies, while Table 5-II provides an overview of the characteristics of the CP specifications. Table 5-II first shows the number of lines of code (LOC) and the size of the CP specification (number of parameters, categories, and choices). The table shows the different types of constraints involved in the CP specifications (next three columns): e.g., the Triangle CP specification has three single choices, three error choices and nine choices that have a constraint with other choices (typically to prevent infeasible combinations of choices). The next seven columns provide data on the latter constraints: for Triangle out of the nine choices, three choices are involved in one constraint and six choices are involved in four constraints. The next five columns provide information on

the contents of the CNF representations of these constraints between choices: e.g., in Triangle, the CNF expressions are made of a total of 12 terms, of which nine terms have two tuples and three terms have three tuples. The last two columns provide information about the percentage of constrained choices out of the total choices and constraints per choice. The information about the constraints among choices is obtained from the representation used for CASA, an example of which is discussed in section 4.3.5.

Given that, for Triangle, choices tagged as Single also have a constraint with other choices (e.g., the three choices tagged as Single are also part of the nine which have a constraint with other choices), the total number of constrained choices is 12 out of 18 (66%). This is not the case for NextDate, which has 13 of its 18 choices constrained (72%). In PackHexChar, 30 out of 32 choices are involved in constraints. This is the most complex academic case study where approximately 94% of choices are constrained (second to the last column in Table 5-II). The last column in Table 5-II gives the number of constraints per choice e.g., triangle has 1.5 constraints per choice.

Although simple in size, for instance in terms of LOC or in terms of the size of the CP specification, these case studies exhibit numerous and complex constraints: 66% (Triangle), 72% (NextDate) and 94% (PackHexChar) of the choices are constrained; constraints not only involve pairs of choices but up to 6-tuples of choices. If we focus only on the constrained choices we observe that in Triangle, 12 choices are involved in 18 constraints (error + single + CNF terms) in an attempt to generate feasible combinations thus leading to an average of 1.50 constraints per choice; choices are involved in up to four constraints and there are up to 3-tuple constraints. In NextDate, 13 choices are involved in 11 such constraints projecting 0.84 constraints per choice; choices

are involved in up to three constraints and there are up to 3-tuple constraints. PackHexChar has a complex CP specification in which 30 choices (error choices and choices involved in constraints) are involved in 30 constraints (error constraints and CNF terms) demonstrating one constraint per choice. An analysis of the PackHexChar CP specification shows choices are involved in up to seven constraints and there are up to 6-tuple constraints making it the most complex of the academic case studies. We note that although a choice may not be explicitly constrained, it may be indirectly constrained because of other constrained choices in the category. While deriving the information about the constraints in Table 5-II, we are counting the choices that are directly involved in the constraints with reference to the choice on which the constraint is specified. Needless to say, that the choices which are not specified in constraints are also indirectly constrained: e.g., if there are two choices *a* and *b* in a category and an explicit constraint on choice *a* specifies that it cannot be combined with choice *c* of another category; this indirectly means that *c* will have to be combined with *b*, thereby indirectly constraining choice *b*.

5.3.2 Characteristics of CP specifications of Level 1 Industrial case studies

We made 10 CP specifications to account for the eight product files for the Level 1 case studies. In the following sections, we discuss the size of the CP specification of each case study without delving into the technical details of the functionality. The details of the CP specifications of the industrial cases studies are in Appendix C.

5.3.2.1 Category partition specification for CS_ L1_A

Table 5-III gives an overview of the Category Partition specification for this case study. It is a small functional unit belonging to a bigger product file with about 65 LOC.

The initial test suite used 11 parameters of the functional unit in its tests and, analyzing this functionality, we instead obtained five parameters and 15 choices by using CP. The table shows the different types of constraints involved in the CP specification (next three columns): there is one single annotation, no error annotation. Furthermore, one single and another 11 of total 16 choices are involved in constraints. Out of these 11 choices, seven are involved in one constraint whereas one choice is involved in a maximum of four different constraints.

The last set of columns provide information on the contents of the CNF representations of these constraints between choices (a format required by CASA [20] to produce test frames with Melba [10, 11]). Akin to our assumption for academic case studies, we are counting the choices that are directly constrained and are specified in the constraint itself. For this case study, the total number of constrained choices is 12 out of 16 (75%). If we focus only on the constrained choices, we observe that 12 choices are involved in 12 constraints (constraints among choices + single) projecting 1.0 constraint per choice and there is a maximum of three-tuple constraints.

5.3.2.2 Category partition specifications for CS_ L1_B

This is the biggest of the entire product files in level 1 with approximately 1655 LOC (Table 5-III). The function in this file calls 12 other functions and the testing of the whole unit is achieved both using manual and simulator based testing. The original test suite had 29 parameters of the functional units in its tests with different values and eight environment variables, which were insufficient for testing this product file. Using CP, we introduced nine more parameters making the number of parameters 46.

This CP specification does not employ many constraints. Out of 106 choices, only 15 are involved in constraints. 14 of the 15 choices are involved in one constraint and one is involved in two constraints. The total number of terms in the CNF (constraints) is eight where seven are involved in two tuple constraints and one is involved in a 3-tuple constraint. Since 15 choices are involved in eight constraints, there are 0.53 constraints per choice.

5.3.2.3 Category partition specifications for CS_ L1_C

Table 5-III shows the characteristics of the CP of case study L1_C. This CP specification corresponds to a medium size product file and the original test suite contains both manual and simulator based tests. Since this product file corresponds to a complex functionality, we employed a number of constraints among choices along with one single annotation. The initial test suite had nine parameters of the functional unit in its tests but according to the functionality of the product, we decided to choose five

Table 5-III: Characteristics of CP specifications of CS_L1_A, CS_L1_B and CS_L1_C

							Total Choices involved in X constraints														
Case study	LOC	Parameters	Categories	Choices	Single Choices	Error Choices	X= All constraints	X= 1 constraint	X = 2 constraints	X= 3 constraints	X = 4 constraints	X= 6 constraints	X = 7 constraints	X = 11 constraints	No. of terms in CNF	2-tuple constraints	3-tuple constraints	4-tuple constraints	5-tuple constraints	% of choices constrained	Constraint per choice
CS_L1_A	65	5	5	16	1	0	11	7	1	2	1	0	0	0	11	10	1	0	0	75	1
CS_L1_B	1655	46	46	106	0	0	15	14	1	0	0	0	0	0	8	7	1	0	0	14	0.5
CS_L1_C	329	6	6	21	1	0	11	0	6	1	1	1	1	1	12	2	2	6	2	52	1.18

parameters and one environment variable.

Out of 21 choices, 11 are involved in constraints, therefore 52% of the choices are constrained, and these 11 choices are involved in 13 constraints that is approximately 1.18 constraint per choice. This is a highly constrained case study because there are few choices that are involved in seven or up to 11 constraints. Additionally the size of the constraints is large with six 4-tuple constraints and two 5-tuple constraints.

5.3.2.4 Category partition specification for CS_L1_D to CS_L1_J (Seven Level 1 Case Studies without constraints)

In this section, we discuss the characteristics of category partition specifications of those case studies which do not employ constraints among the choices, i.e., CS_L1_D, CS_L1_E, CS_L1_F, CS_L1_G, CS_L1_H, CS_L1_I, CS_L1_J. The characteristics of all the case studies are shown in Table 5-IV. Using the CP technique, we reduced the use of the original nine parameters, which were used to test the functionality of the functional unit in the test cases, to a use of six parameters for the case study L1_D. Further, there are six categories and 19 choices in this CP specification. There is no constraint among the choices and no single and error annotations. We also observed that the original test

Table 5-IV; Characteristics of case studies without constraints (D, E, F, G, H, I, J)

Case study	LOC	Parameters	Categories	Choices
CS_L1_D	342	6	6	19
CS_L1_E	112	3	3	6
CS_L1_F	68	2	2	6
CS_L1_G	72	5	5	12
CS_L1_H	92	5	5	22
CS_L1_I	900	4	4	13
CS_L1_J	21	1	1	3

suite, besides having nine parameters, missed some important values for those parameters which resulted in incomplete testing and lower code coverage. In case study L1_E, the original test suite used two parameters for testing the functionality of the product code but we discovered three parameters on the basis of CP technique. There are six choices and no constraints among choices or single or error annotations. Case study L1_F has six choices corresponding to two categories and no constraints.

In case study L1_G we identified those parameters in the original test suite that were not contributing to the testing of this product code and so reduced the parameters from seven to five. Again there are no constraints among choices or single or error annotations.

The CP specification of L1_H corresponds to two product files. The CP technique resulted in reducing the number of parameters from eight to five. The original test suite lacked testing for all possible values of parameters. Therefore, we introduced more choices based on the functionality of this software unit. In L1_I, the CP technique resulted in reducing the number of parameters from nine to four. Case study L1_J is the smallest of all product files and hence functionality. It contains one parameter one category and three choices formed using boundary value analysis.

5.3.3 Characteristics of CP specifications for Level 2 Industrial case study

The level 2 case study relies on level 1 functionality through function calls in addition to other functionalities that this unit fulfills. There is only one case study in this category: CS_L2_A. Appendix C.11 reports on the CP specification for this case study. We identified 15 categories belonging to 11 parameters and 46 choices. We specified 27 constraints in CNF format to ensure that the choices are combined to form valid test frames.

Table 5-V: Characteristics of CP Specification of CS_L2_A

							Choices involved in X constraints												
Case study	LOC	Parameters	Categories	Choices	Single Choices	Error Choices	X = all constraints	X = 1 constraint	X = 2 constraints	X = 3 constraints	X = 5 constraints	X = 6 constraints	No. of terms in CNF	2-tuple constraints	3-tuple constraints	4-tuple constraints	6-tuple constraints	% of choices constrained	Constraint per choice
CS_L2_A	1060	11	15	46	0	0	39	21	8	5	3	2	27	13	5	7	2	85	0.7

This case study is one of the highly constrained industrial case studies in which 39 choices out of 46 are constrained, i.e., 84.7% of the choices are constrained. The complexity of the constraints is apparent from the fact that 21 choices are involved in one constraint and eight, five, three and two choices are involved in two, three, five and six constraints respectively. Although the number of constraints per choice is small, i.e., 39 choices are involved in 27 constraints (0.7 rate), we encountered two 6-tuple constraints which shows a maximum of six choices are constrained together, similar to PackHexChar case study. Table 5-V contains other details.

5.4 Test frame construction

The next step after creating the CP specification is the selection of covering array generation tools for the generation of test frames. We used three tools for the generation of test frames. We selected CASA [20, 53], ACTS [19] and PICT [21] from the survey described in Chapter 4. We had searched for the best alternative covering array generation tools to create test frames and identified that these three tools were adequate [16]: e.g., they are freely available, they support full-fledged constraints. Since these three tools rely on different techniques we asked ourselves whether the teachnique employed to create test frames would have an impact on the "quality" of the

corresponding test cases. Specifically, CASA uses a metaheuristic technique (Simulated Annealing), PICT uses a greedy algorithm, following a Test Based Generation strategy, and ACTS (IPOG) uses a greedy algorithm following a parameter based generation strategy [16].

CASA and ACTS are used with all the academic (Triangle, Next Date and PackHexChar) and industrial case studies (Level 1 and Level 2) whereas PICT is used for only one industrial Level 2 case study. We used PICT mainly to compare the cost and quality (code coverage) of the test suite generated by a greedy test based algorithm with the greedy parameter based algorithms and metaheuristic algorithms. We used the Level 2 case study because this is the largest of all the case studies and we assumed that the results would be clearly distinguishable.

CASA [53] is integrated with Melba [10, 11], to generate t-way covering arrays. Melba [10, 11] is an automated tool which accepts CP specifications and the constraints among the choices along with single and error annotations and presents the results to CASA in a format it can accept. For the experiments based on ACTS and PICT, the input is provided manually to these tools in an appropriate format. We proceeded manually with ACTS and PICT mainly because we were using them for a limited number of experiments and we wanted to have an initial idea of their performance when used to support CP prior to spending time to integrate them into Melba.

Since the notions of error and single choices are not known to the techniques for producing covering arrays (e.g., CASA), Melba implements a specific procedure to handle these situations. The same procedure is adopted while manually using ACTS for

the experiments; our use of PICT does not require the procedure as the Level 2 case study has no single annotation. The procedure is detailed below.

When integrated with CASA, Melba first calls CASA without single and error choices (they are removed from the CP specification) in the set of constraints and saves the resulting test frames. Melba then executes CASA again with the complete CP specification but omitting the error choices. Melba then keeps the first test frame that involves each single choice in the set of test frames obtained from CASA: a single choice should appear only once in the set of test frames. Last, we used the Single Error criterion [13] to add test frames involving error choices. This ensures that each error choice is in exactly one test frame and is combined with other non-error choices of other categories. Melba then adds each test frame for error and single choices in the earlier saved set of test frames to obtain the final test suite.

The test frames are generated using different selection criteria. We have used Each Choice (EC), Base Choice (BC) and its proposed extensions CBC and ECBC (Chapter 8), Pair-Wise (PW) and 3-way (Triplet). In our experiments, test frames were generated manually for CBC and ECBC and automatically for the combinatorial criteria, i.e., EC, BC, PW and Triplet using Melba [10, 11], our tool support for CP which interacts with CASA [53] and ACTS [19] and PICT [21].

In this thesis, we have designed a number of experiments to gauge the impact of constraints (single, error and constraints among choices) on the quality of test suites. The covering array generation tools facilitated those experiments. These tools helped by automating the process of test frame generation when constraints among choices or single and error annotations were ignored and when they were considered. The next section

discusses the methodology that we used to obtain test inputs and hence test cases when these constraints were ignored, knowing that ignoring the constraints can lead to infeasible test frames.

5.5 Generation of test cases on the basis of test frames

The next step is to find test inputs for the parameters that satisfy each test case specification (i.e., test frame). We begin by discussing the principles that we followed for the test input selection when the test frames are feasible, because of the presence of constraints among choices, that forbids infeasible choice combinations or when the test frames are infeasible because we ignored those constraints among choices because of the experiments that we conducted.

5.5.1 Test input selection principles

The test input selection while accounting for constraints among the choices would lead to feasible test frames. Therefore finding test inputs for such test frames is simple and systematic. We manually assigned values to the choices while making sure that the choice specifications are satisfied and at the same time making sure that the constraints among the choices are satisfied. We will discuss the test input selection of each case study separately in sections 5.5.2, 5.5.3 and 5.5.4.

In our experimental work, we also investigate the impact of the absence of constraints among choices thereby resulting in infeasible choice combinations because they contain choices which specifications contradict one another. (Recall that choices implicitly or explicitly specify equivalence classes for input values.) We nevertheless want to use those test frames and therefore need to find test inputs, recognizing that those

inputs will satisfy some specifications of the choices in the test frame but not other choice specifications.

When constraints are discarded, many test input selection strategies can be considered. For instance, reading choices from left to right one can satisfy their specifications in that order; if, along the way, finding an input to satisfy a choice specification is infeasible because of decisions already made in order to make previous choice specifications feasible, we ignore that choice specification and proceed to the next choice in the test frame. This option has a limitation. The choices in the CP specification are structured in the order of appearance of parameters. This option would always give higher priority to the constraints of the first parameters, which would therefore always be satisfied. This would introduce a bias in the experimental results.

We instead adopted a priority model based strategy that is reproducible and is followed for the academic case studies.

In case of a test frame that is infeasible because of conflicting choice specifications, we propose a **priority model** that ranks choices in the test frame. A choice specification specifies the scaffold under which a value can be assigned to a choice. This scaffold can be a general permitted range (e.g., $x<5$, $y>5$) or can depend on the values of other parameters (e.g., $x>y$, where x and y are parameters). The general principle is to give a high priority to choices whose choice specification can be easily satisfied and lower priority to choices whose choice specification is hard to satisfy. This being said, if a choice specification is independent of the values of other parameters it will be easier to satisfy and hence will be assigned a higher priority than when it is dependent on other parameters. In the latter case, satisfying the choice specification will also depend on the

values that were assigned to those individual parameters, which will make the satisfiability hard to accomplish. For instance, it is impossible to satisfy x>y if we have already decided to satisfy x<5 and y>5.

Therefore, we give higher priority to those categories and hence choices whose choice specification do not depend on any other parameter other than the parameters of its own category. This is followed by giving a lower priority to those categories, hence choices, whose choice specification exhibit higher dependence on other parameters. For instance, we assign first priority class to those categories and hence choices whose choice specification do not refer to any other parameter. Similarly, we assign to second priority class to those categories that depend on one other parameter and so forth. Following this process, we will have each category (and therefore its choices) assigned to a specific priority class. We then assigned test inputs while satisfying the choice specifications in the order of their priority. It is always possible to satisfy the choice specification of the choices which belong to the first priority class, even for the infeasible test frames. For the other priority classes it gets difficult to satisfy the choice specification due to the decisions we make for the higher priority classes. Therefore, the test inputs for the infeasible test frames will largely depend on the choice specifications belonging to the higher priority classes than those of the lower priority classes.

5.5.2 Test input selection for the Triangle case study

For the test frames involving constraints among choices, we selected the value for the sides of the triangle in their acceptable ranges. The Triangle CP specification has three error choices for out of bound values. When a test frame involves one of these choices, we systematically selected a null (0) value. Other choice specifications relate to

Table 5-VI: Priority classes for triangle case study

Priority classes	Categories of Parameter A	Categories of Parameter B	Categories of Parameter C
PC1	Value of SA (Ch1.1, Ch1.2)	Value of SB (Ch4.1, Ch4.2)	Value of SC (Ch7.1, Ch7.2)
PC2	SA compared to SB (Ch2.1, Ch2.2)	SB compared to SC (Ch5.1, Ch5.2)	SC compared to SA (Ch8.1, Ch8.2)
PC3	SA compared to SB and SC (Ch3.1, Ch3.2)	SB compared to SC and SA (Ch6.1, Ch6.2)	SC compared to SA and SB (Ch9.1, Ch9.2)

triangle inequalities. When selecting input values for other test frames, we made sure those inequalities were satisfied and we selected random values in permitted ranges.

For the infeasible test frames, we used the priority model discussed previously (section 5.5.1), which is summarized in Table 5-VI. We assigned categories in the CP specification to different priority classes according to how much the category specification refers to other parameters as shown in Table 5-VI. The first priority class has categories which specification does not refer to any another parameter than this category's parameter. (As a result the specifications of the choices in such a category do not refer to any other parameter.) The second priority class has categories whose specification refers to only one other parameter. The third class has categories whose specification refers to two other parameters. Choice specifications for the selection of test inputs are then satisfied in increasing order of priority. This approach resulted in satisfying all the choice specifications in the first priority classes while choices specifications in other priority classes may not be satisfied since test frames may not be feasible. This process applies to each test suite. Given that the Triangle CP specification has at least one category of priority one for each parameter this procedure ensures that we are always able to find test inputs even though a test frame may have contradicting choice specifications.

5.5.3 Test input selection for the Next date case study

The NextDate CP specification has six error choices specifying invalid input values (dates). When a test frame involves an error choice for the date or month, we selected value 0 for the lower out-of-bound value and 13 (month) and 32 (day) for upper out-of-bound values. When a test frame involves an error choice for the year, we selected 1581 and 2101 as lower and upper out-of-bound values for a common year and 1580 and 2104 as lower and upper out-of-bound values for a leap year. For other test frames the test inputs are selected in a strategic manner. Starting from the first permitted input value for the day, the month or the year, the value is incremented every time for each new test case modulo the maximum value allowed by the choice specification in the test frame. If a choice specification is a disjunction, e.g., there are four 30 days months, we selected each value in a round-robin manner. This process applies to each test suite. The nature of the NextDate CP specification is such that each category specification only refers to that category's parameter; reusing the notion of priority from the previous section, all parameters are in priority one. This means that it is always possible to find test inputs even when a test frame is obtained by omitting constraints.

5.5.4 Test input selection for the PackHexChar case study

The PackHexChar case study has four error choices corresponding to out of bound values for input variables RLEN and ODD_DIGIT. When a test frame involves these error choices, we used the same test case in all the test suites irrespective of the selection criteria to ensure that the error test cases perform similarly in all the test suites. For instance, when the value of RLEN is less than zero the test case is RLEN =-1, ODD_DIGIT= -1 and Input String = "abc", i.e., (-1, -1, "abc") and we used this test case

in Each choice, Pairwise or Base Choice adequate test suites. Similarly, when RLEN is greater than the string length the test case is (6, -1, “abc”). When ODD_DIGIT is less than -1 the test case is (2, -2, “abc”) and for greater than 15 the test case is (2, 16, “abc”).

For the other test frames, the test inputs are selected such that all the constraints are satisfied. The test inputs corresponding to the choices of the ODD_DIGIT parameter are selected in the permitted range by incrementing one value at a time. For example, for hexadecimal characters *a* to *f*, the values are selected in a round robin fashion and the same process is followed for each test suite.

We selected the test inputs for RLEN based on constraints and its combinations with other choices. For the test frames where the value of RLEN is expected to be less than or equal to the input string, we varied the value of RLEN from 1 to 7 randomly among the test cases in a test suite. We ensured that each test suite has the RLEN value of 1 when the length of RLEN is less than the input string length. We combined this value of RLEN when the number of hexadecimal characters in string is “odd” because it is not possible to have this combination when the number of hex characters is even (appendix B.3).

We selected the Input String on the basis of the CP specification and the constraints between various choices in the CP specification. In case of a non-hexadecimal input string, we generally selected the string “sunint”. We further varied the size and type of the string based on the RLEN value and the number and type of hexadecimal characters required in the string (based on the combination of choices). We ensured that the test suites have test cases with strings containing all hexadecimal alphabets (upper case and lower case) and numbers (0 to 9) to ensure completeness. A string would begin with a (or A) if the string is a solely hexadecimal string containing lower (or upper) case alphabets

and would contain all hexadecimal characters until f (or F). If the string is a numerical string, it contains numbers from 0 to 9 in an increasing order. If the string is a mixed string containing hexadecimal and non-hexadecimal characters, we selected hexadecimal characters in the same way as above but inserted non-hexadecimal characters in the string. We ensure that each test suite has all hexadecimal numbers and characters in the input string. In this process of input string selection, we however did not make a check if all the RLEN substrings have all the hexadecimal characters and hexadecimal numbers. We observed minor deviations in the mutants being killed because of that; we will detail this discussion in section 9.3.3.

5.5.5 Test input selection for Industrial case studies

In the Level 1 and Level 2 industrial case studies, most of the parameters take discrete values. Therefore, this leads to categories decomposed into choices that each represents a unique input value: one choice per discrete value. This is similar to a GUI function with an integer input parameter that takes specific values representing colours: 0 for black, 1 for bleu etc. This leaves very little room to change input values. Other parameters have choice specifications with large ranges; we formulated equivalence partitions and selected test inputs based on boundary values and randomly on values in their permissible limits.

We further encountered situations where we had to select test inputs in the presence and absence of constraints.

The input selection while accounting for constraints among the choices is simple and systematic. We assigned values to the choices, systematically, while making sure that the

choice specifications are satisfied and at the same time making sure that the constraints among the choices are satisfied.

When discarding constraints among choices, some test frames will necessarily become infeasible. We followed the priority model we discussed in section 5.5.1. We observed that most of the choice specifications for the industrial case studies do not depend on any other parameter than the parameter of its own category. Therefore, most of the choices belong to the first priority class. It is because of this reason it was possible to obtain test inputs for the test frames even in the absence of constraints.

5.6 Measurement

As mentioned earlier, we are interested in the "quality" of generated test suites. Once test inputs and oracles are identified for test cases, they are executed and test suite quality is measured. Similar to many test experiments reported in the literature, we used surrogate measures of quality, i.e., structural coverage of the source code, fault detection effectiveness, data flow coverage and number of test frames/cases. The NextDate case study is implemented in Java as two classes where the Date.Java class is extended as the NextDate class. The NextDate class implements all the functionalities related to finding an invalid month, year and date and then finding the next date. We will report on the measurements for the NextDate class, since it has most of the functionalities. The Triangle case study is implemented as Triangle.Java and PackHexChar is implemented as PackHexChar.java, so we report on the measurements for these classes.

For the industrial case studies we used coverage of the source code and test cost to estimate the quality of test suites. However to obtain test frames and hence coverage and

cost and to draw assertive conclusions we used two tools (CASA and ACTS) and PICT for Level 2 case study.

5.6.1 Control Flow Coverage

For the academic case studies we used Codecover (www.codecover.org) to measure different types of coverage namely statement coverage, branch coverage, loop coverage and term coverage. A 100% statement coverage ensures that each statement of the program has been executed at least once. 100% branch coverage ensures that each branch has been executed at least once. A branch corresponds to both if and else parts of a branching statement (e.g., if-then, if-then-else, loop). Loop coverage ensures that the loops are carefully tested and checks the result by omitting the loop, executing the loop once and multiple times. Term coverage is a flavour of MC/DC [111] coverage. It ensures that each Boolean term in the Boolean expression of a branching statement affects at least once the overall result of the Boolean expression.

We used Cobertura (cobertura.sourceforge.net) to measure statement coverage for the industrial case studies. This was the tool permitted with the proprietary software.

5.6.2 Fault detection effectiveness

We used mutation analysis to evaluate fault detection effectiveness of a test suite. A mutant is a fault which is seeded in the program to obtain a mutated version of the code. A mutant is killed by the test case when its behavior (e.g., classification of a triangle, returned next date) differs from the original, un-mutated program for the same inputs. The effectiveness of a test suite is measured as the percentage of mutants it kills, called the mutation score. We used MuClipse [112] to automatically generate mutants. MuClipse is an Eclipse plugin for the MuJava mutation engine. The plugin is a user-

friendly interface for generating the mutants and for running mutated version against JUnit test cases.

MuJava [113] is a Java based tool that automatically generates mutants using different types of mutation operators. MuJava uses two types of mutation operators: Class level mutation operators [114] and method level mutation operators [115]. A class level mutation operator creates mutants corresponding to object oriented features. They are classified into four groups namely encapsulation, inheritance, polymorphism and some java specific features. Method level mutation operators are the operators used in a method of a class. These operators modify the expressions by replacing, deleting and inserting primitive operators. Various categories of method level mutation operators are arithmetic, relational, conditional, shift, logical and assignment operators. We used all the mutation operators available with MuJava. The complete list of those operators is in Appendix D. Mutants have shown to be representative of real faults for similar experiments [116].

MuJava generated 221 mutants for the Triangle case study. There are 113 mutants generated for the NextDate class of the Next Date case study. Since we are reporting on the code coverage of one class out of the two classes in the NextDate case study we created mutants only for this class. For the PackHexChar class MuJava generated 643 mutants.

In the process of mutation analysis there can be a mutant, which would show behaviour identical to the original program regardless of the input values and is not killed by any test case. These mutant is called equivalent mutant. There are heuristics and studies to identify equivalent mutants either by the process of automation or by dealing with them manually. Either way there are studies in support of the fact that identifying

equivalent mutant is an undecidable and resource intensive process [117, 118]. However a commonly used heuristic to identify equivalent mutant is to consider those mutants not killed by any test case as equivalent [119]. In this thesis, we will consider this heuristic for identifying equivalent mutants. In this thesis, we are conducting a number of experiments using more than one test frame generation technologies and more than one selection criteria. This results in a large number of test suites. If all the test suites have a definite number of mutants that are not killed by any test case then it is reasonable to consider those mutants as equivalent mutants. With such a large number of test suites, the chances of making an incorrect selection of equivalent mutants are reduced. We will however discuss the mutants and the reasons why they remain alive wherever necessary.

MuClipse created 221 mutants for Triangle case study, 27 of which are not killed by any test suite. Three mutants out of these 27 can be killed if the values of the three sides of the triangle are less than zero. We however systematically selected 0 as an out of bound value for triangle sides. Modifying the CP specification to include both zero and less than zero values separately can lead to killing these mutants. Another three mutants out of 27 can be killed if different sets of test inputs are chosen than the current ones, for instance test inputs for an isosceles triangle. While selecting test inputs for an isosceles triangle we are making a random selection for the sides of the triangle (a, b, c), e.g., (3, 3, 5) or (5, 5, 6). But while analysing the killed mutants we observed that the random selections that we made had an impact on the mutants that were being killed: i.e., (3, 3, 5) killed the mutant b++ (arithmetic operator mutant) whereas (5, 5, 6) did not (after incrementing b in 3,3,5 the triangle is a scalene triangle but in 5,5,6 triangle continues to be an isosceles triangle). Therefore making such selections have an impact on a subset of

mutants out of 221. This also shows that equivalence classes and boundary values, even embedded in a structured way in CP, are not a panacea. One scenario of such an impact is discussed in section 9.3.2.3. The remaining 21 mutants are equivalent mutants, which we confirmed manually.

MuClipse created 113 mutants for NextDate, 24 of which are not killed by any test suite. Out of these 24 mutants, 16 mutants are equivalent mutants (they have the same behaviour as the original program and hence are not killed by any test case). Out of the remaining eight, six mutants are not killed because they belong to the part of the code where Feb has more than 29 days. As per our CP specification, this combination is infeasible resulting in no test frame: a larger than expected number of days in a month is a condition that can be exercised as per the CP specification but not with the month of February. The remaining two mutants are alive because we do not have a test input combination for Dec 31, 2100. This is because we have chosen the test inputs in a round robin fashion; starting from 1582, we are incrementing the year one at a time so we did not reach year 2100 as a test input.

MuClipse created 643 mutants for the PackHexChar case study. 110 mutants out of 643 are not killed by any of the test suites. 34 mutants out of 110 belongs to the *equals()* function, which compares two PackHexChar objects. The CP specifications for this case study focuses on the PackHexChar functionality, i.e., packing hexadecimal characters in pairs given the values of RLEN and ODD_DIGIT. We did not include comparison of PackHexchar objects as part of the CP specifications. Therefore, we do not have any test case that focuses on testing the *equals()* function and so the related mutants remain alive. Out of the remaining 76 alive mutants, one mutant is not killed because, based on our CP

specifications, we do not have a test frame where RLEN is less than zero and ODD_DIGIT is a valid hexadecimal number. The value of RLEN less than zero is an error choice and we have fixed the error test cases where this specific choice is combined with ODD_DIGIT value of -1 (no hexadecimal character). The remaining 75 alive mutants are equivalent mutants and they yield the same behaviour as the original program.

We do not report on fault detection effectiveness of Industrial cases studies due to proprietary reasons.

5.6.3 Data Flow Coverage

Data flow testing criteria require that a test case exercises a path in the program where a variable is assigned a value and is subsequently used hence ensuring that the values created at one point in the program are used correctly [1]. The use of the value assigned to the variable can occur in a computation (c-use) or in a predicate (p-use). Both these types of uses collectively are called Definition Use associations (DUAs) and the associated path from a definition to a use is called a Def-Use path [120]. An advantage of data flow coverage is that it helps us analyze how a program handles data along those def-use paths. We will consider data flow coverage as one of the measures to evaluate the quality of test suites.

We used Ba-Dua [121, 122] for measuring the data flow coverage for the test suites on the academic case studies. Ba-Dua (Bitwise Algorithm – powered definition-use association coverage) is a data flow-testing tool to measure data flow coverage for Java programs that uses an efficient algorithm during instrumentation of the code to reduce memory usage [120].

The execution of the program using Ba-Dua generates a coverage file, which provides the information about the Def Use (DU) coverage for each class and each method in that class. The Definition Use criterion refers to the definition of a variable and the subsequent uses of that definition. Ba-Dua returns the ratio of the number of covered DUs in a class to the total number of possible DUs within a class. We can therefore see if a test suite has missed specific DUs. The tool also returns the DU coverage per method in a specific class. These results are part of an XML report generated at the end.

We used data flow coverage as a comparison criterion only in Chapter 9 for measuring the results of the proposed criteria i.e., Constrained Base Choice and Extended Constrained Base Choice with the existing criteria like Base Choice, Each Choice, Pairwise and Three Way criteria. We could not use Ba-Dua on the industry case studies due to proprietary reasons.

5.6.4 Cost

The cost of testing a system can depend on the time and resources required for executing the tests [123]. Further, these factors are typically directly proportional to the size of the test suite: the larger the number of tests the more resources will be utilized. We therefore measure, similarly to many others before us (e.g., [11, 124]), the cost of a test suite as the number of test cases (in fact test frames) of that test suite.

In case of the Industrial Level 2 case study, which has a larger and highly constrained CP specification, we also measured the time taken by the combinatorial testing tools to generate the test frames in the presence and absence of constraints using three tools (CASA, ACTS and PICT). The time taken for the execution is measured on a machine with Microsoft Windows 7 Enterprise Edition, Intel Core i7 3.60 GHz processor

and 32 GB RAM. The time is measured using the Unix Time command and is executed on the Windows machine using Cygwin. Cygwin is a freeware that creates a Unix like environment on Windows and lets a windows machine emulate Unix servers [125].

In the case of the industrial case studies, we were constrained due to proprietary reasons to use a specific coverage tool (Cobertura), different sets of experimental criteria (because of the absence of error annotations) and could not perform mutation analysis. Using measurements identical to the original test suite, it was possible to compare the test suite created using the CP technique with the existing (original) test suite and evaluate the improvements.

Chapter 6 —Results of Category Partition implementation on Industrial Case Studies

The general objective of our project at Ericsson Inc. was to assess the suitability of Category Partition testing as a test strategy for testing a 4G LTE middleware. In the original test suite, test cases were obtained either using a simulator or manually. We analyzed all the possible ways a product file was being tested. We analyzed the existing test suites and recorded their measurements (i.e., code coverage, cost). We also inspected the parameters used for testing the code and the values of those parameters used in the test cases. We then proposed test suites using CP, recorded their measurements, and compared them with that of the existing test suites. We compared the number of parameters we obtained using CP with the ones in the original test suite. The next sections discusses the quality of the existing test suite. We then discuss the results of CP implementation and compare the quality of the test suite obtained using CP with that of the original test suite.

6.1 Quality of the original test suites (without Category Partition)

Table 6-I shows the code coverage and cost (number of test cases) for the different product files in Level 1 and Level 2 of the industrial case studies. The table contains the list of all the product files, their corresponding CP specifications (which will be used later in this Chapter), the statement coverage for each product file using the original test suite, the cost associated with each product file in terms of number of test cases. The column where the number of tests are separated by '+' shows those product files that are tested using multiple test files possibly using a combination of different testing techniques

Table 6-I: Coverage and cost (number of test cases) of the original test suite

	Product files	CP specifications	Coverage %	No of test cases	LOC
1.	Product file 1	CS_L1_F	80	1	70
2.	Product file 2	CS_L1_C	85	560+24	300
3.	Product file 3	CS_L1_A and CS_L1_B	83	2+2+72	1655
4.	Product file 4	CS_L1_H and CS_L1_I	99	27+560	900
5.	Product file 5	CS_L1_J	100	1	15
6.	Product file 6	CS_L1_D	100	560+20	340
7.	Product file 7	CS_L1_E, CS_L1_G, CS_L1_H	100	1+3+27	180
8.	Product file 8	CS_L1_E	100	1	105
9	Product file 9	CS_L2_A	86	225	1060

(manual and simulator based testing). We also analyzed the current test suite to see whether the parameters currently used are sufficient for testing the code or if there is an overuse or underuse of parameters. The table also contains the size of the product file as lines of code.

6.2 Results of implementation of CP on level 1 case studies

We first discuss code coverage and cost achieved by implementing each CP specification on a full product file or a part of it as structured in Table 5-I. Thereafter, we discuss the measurements for each product file and compare it with the original suite Table 6-I.

6.2.1 Result of CP implementation for CS_L1_A

This CP specification implements a subset of functionalities of product file 3. The code coverage for the whole file is 83% (Table 6-I) but this subset of functionality which we envisioned as CS_L1_A has the initial coverage of 66% using 11 parameters. We analyzed the parameters for their contribution in testing this functionality and observed an overuse of parameters. We further analyzed that this large number of parameters was not able to achieve adequate coverage (66%) and narrowed down the problem to

Table 6-II: Result of CP implementation of CS L1_A using CASA and ACTS

	Selection Criterion	Non Constrained (-C, -S)		Constrained (+C, +S)	
		Cost	Coverage %	Cost	Coverage %
	Initial test suite	2	66	NA	NA
CASA	Base Choice	12	88.8	9	88.8
	Each Choice	6	88.8	10	100
	Pair Wise	18	100	11	100
ACTS	Base Choice	12	88.8	NA	NA
	Each Choice	6	100	9	100
	Pair Wise	18	100	12	100

improper selection of parameters. We created a CP specification with five parameters and 16 choices (Table 5-III). The results of the CP implementation and the quality of the test suite are shown in Table 6-II using CASA and ACTS. The original test suite does not implement constraints among the choices nor does it implement single or error annotations. Therefore, while using CP we monitored the results in the absence and presence of constraints among choices to gauge the improvements of CP implementation in the incremental manner and to be able to better compare with the original test suite. The third and fourth columns correspond to results when constraints among the choices and single annotations were ignored (-C, -S) in the CP specification but parameters, categories and choices were used and combinations among the choices were made as required by the CP technique. We then again generated test cases using the CP specification while accounting for constraints among the choices and single annotations (+C, +S) in columns five and six. The Base Choice criterion is implemented in Melba in the presence of constraints among choices therefore we have a value for it in Table 6-II whereas ACTS does not have the functionality of generation of test frames using the Base Choice criterion in the presence of constraints; as a result we do not have a value for it in the table.

We observe that CP consistently improves the quality of the test suite: higher coverage than initial test suite. A mere implementation of CP without even implementing the constraints has improved the coverage from 66% to 100% using Each Choice and ACTS test frame generation.

We observe that ACTS, which is deterministic in nature, gives 100% coverage both in the presence and absence of constraints. An Each Choice adequate test suite generated by CASA however does not result in 100% code coverage in the absence of constraints, which can be attributed to the non-deterministic nature of CASA. Subsequent executions of CASA can lead to the selection of different choices and hence results similar to ACTS but for our experiments, we are relying on the first execution of CASA. We observed for both ACTS and CASA that in the presence of constraints (+C) and for the Pairwise criterion the cost reduces by approximately 33% when compared to an absence of constraints (-C), unlike Each Choice criterion where the cost increases by approximately 40%. The reason for the observation is detailed in section 7.4.3.

6.2.2 Result of CP implementation for CS_ L1_B

This is the biggest of all product files among the industrial cases studies. We introduced nine more parameters to the earlier 37 to account for complete testing. We also introduced more values for a few parameters. The initial test suite with 74 test cases led to 81% coverage of this product file. With the CP specification, with 46 parameters, we generated five test cases using Each Choice, 61 using Base Choice and 24 using Pairwise (Table 6-III). Using five Each Choice adequate test cases we achieved the coverage of 80%, which is a 93% reduction in the size of the test suite from the initial test suite for the same coverage. We did not implement the test suite obtained using the other

Table 6-III: Result of CP implementation of CS_L1_B using CASA and ACTS

	Selection Criterion	Non Constrained (-C)		Constrained (+C)	
		Cost	**Coverage (%)**	**Cost**	**Coverage (%)**
	Initial test suite	74	81	NA	NA
CASA	**Base Choice**	61	*	61	*
	Each Choice	**5**	*	**5**	**80**
	Pair Wise	25	*	24	*
ACTS	**Base Choice**	61	*	61	*
	Each Choice	**5**	*	**5**	**80**
	Pair Wise	28	*	28	*

selection criteria due to the large number of parameters and time constraints imposed on this project. Recall also that part of the testing activities, especially the selection of test input values, was manual, which was not convenient on large test suites we obtain for this case study. This calls for some work on automating the transformation of test frames (i.e., test case specifications) into test cases. However, we are certain that implementing other selection criteria (Base Choice and Pairwise) would certainly lead to an increase in coverage because of more combinations.

We observe that since this case study is not highly constrained, i.e., 15 out of 106 choices are constrained (Table 5-III), introducing constraints has resulted in reducing the number of test frames by one when using CASA. Otherwise, results are identical. CASA has also generated fewer test frames for Pairwise as compared to ACTS both in the presence and in the absence of constraints, asserting that the simulated annealing algorithm used by CASA is cheaper than the greedy algorithm used by ACTS in terms of cost but this comes with an expense in the generation time by the algorithm. Table 6-VI further shows the difference in time and cost by both tools.

6.2.3 Result of CP implementation for CS_ L1_C

Table 6-IV: Result of implementation of CS_L1_C using CASA and ACTS

	Selection Criterion	Non Constrained (-C, -S)		Constrained (+C, +S)	
		Cost	Coverage %	Cost	Coverage %
	Initial test suite	560 (Simulator)	51	NA	NA
		24(Manual)	85	NA	NA
CASA	Base Choice	13	82	9	74
	Each Choice	6	79	7	77
	Pair Wise	30	95	26	100
ACTS	Base Choice	13	82	NA	NA
	Each Choice	6	74	8	92
	Pair Wise	30	100	27	100

Table 6-IV shows the results of implementation of CP to product file 2 using CASA and ACTS. This product file renders a coverage of 85% with the initial test suite containing 584 tests. The original suite missed some important parameters and values required for testing this functionality that resulted in low coverage. Using the CP technique, we reduced the number of parameters to five and added the missing choices. The CP specification also contains constraints among the choices and one single annotation (Table 5-III).

Using the Base Choice criterion, we observed a drastic reduction in the number of test cases but there is no improvement in coverage. The coverage in the absence of constraints was better than coverage in the presence of constraints. Base Choice in the presence of constraints implemented using Melba does not subsume Each Choice because of which the coverage in the presence of constraint is less than with the Each Choice criterion. We further observed the coverage for combinatorial criteria (Pairwise), which shows interesting results. For the test frames generated using CASA, in the absence of constraints, 30 test frames were obtained leading to 95% coverage. This is 10% improvement in the coverage from the original test suite at a much smaller cost. When we

generated test frames while accounting for constraints among the choices, the cost reduces by 13% but the coverage increases to 100%. This shows that a constrained CP specification leads to a reduction in cost and an increase in coverage thereby improving the quality of the test suite.

Test frames generated using ACTS showed even better results. We obtained 100% code coverage both in the presence and in the absence of constraints for Pairwise. However, the cost with ACTS is more than with CASA again asserting that greedy algorithms are costlier than simulated annealing technique for the generation of test frames. We also observed that the introduction of constraints has led to increased cost for Each Choice but has reduced the cost for Pairwise for both tools.

6.2.4 Result of CP implementation for CS_L1_D to CS_L1_J (Seven Level 1 Case Studies without Constraints)

In this section, we discuss the results of implementation of the seven Level 1 case studies that do not employ constraints among the choices. The results of the case studies are compiled in Table 6-V.

For the CS_L1_D case study, as compared to the original test suite which has a total of 580 test cases obtained from two simulator based tests suites, CP results in just six test cases using Each Choice with a coverage of 100%: Table 6-V. There is however more test cases using Base Choice and Pairwise (14 and 24, respectively). While analyzing the nine parameters used in the original test suite, we also observed on the basis of the functionality of the product that three of the parameters were not contributing to the testing of the code. Removing surplus parameters results in reducing the number of test frames particularly if the selection criterion is based on combinatorics. Therefore, the

implementation of six test cases using five parameters using Each Choice led to 100% code coverage. If we compare the simulator test suite 1 with the CP generated test suite we observe that CP results in increased code coverage (93% to 100%) at a lower cost (20 to six test cases). This is a 70% reduction in cost while 7% increase in code coverage.

CS_L1_E is a small product (Table 5-IV) where the initial test suite had a code coverage of 100% using one test case and two parameters: Table 6-V. We observed the functionality of the product and introduced one more parameter for completely testing this functionality. This increased cost of the test suite to two with Each Choice while

Table 6-V: Result of CP implementation for CS_L1_D to CS_L1_J

	Selection Criterion	Cost	Coverage
CS_L1_D	**Initial test suite**	20 (simulator test suite 1)	93
		560 (simulator test suite 2)	100
	Base Choice	14	100
	Each Choice	**6**	**100**
	Pair Wise	24	100
CS_L1_E	**Initial test suite**	1	100%
	Base Choice	4	100
	Each Choice	**2**	**100**
	Pair Wise	6	100
CS_L1_F	**Initial test suite**	1	100%
	Base Choice	5	100
	Each Choice	**2**	**100**
	Pair Wise	9	100
CS_L1_G	**Initial test suite**	3	100%
	Base Choice	8	100
	Each Choice	**3**	**100**
CS_L1_H	**Initial test suite**	27	100
	Base Choice	18	100
	Each Choice	**6**	**100**
	Pair Wise	36	100
CS_L1_I	**Initial test suite**	560	97%
	Base Choice	10	--
	Each Choice	**6**	**97%**
	Pair Wise	18	--
CS_L1_J	**Initial test suite**	1	100%
	Each Choice	3	100%

maintaining code coverage to 100%.

In CS_L1_F, although the number of parameters is the same for all the test suites, i.e., 2, we observe that the original test suite does not have test cases with all possible choices. Therefore, we introduced choices based on boundary value analysis to ensure completeness. The number of parameters is the same as the original test suite. We obtained 100% code coverage for all the selection criteria in a cost greater than the initial test suite mainly because we were using combinatorial testing tools.

Based on the functionality of CS_L1_G, we reduced the number of parameters from seven to five when compared to the initial test suite (Table 6-V). We also introduced a few more choices to ensure completeness in testing, which resulted in a cost increase while maintaining the coverage to 100%.

In CS_L1_H, the initial test suite employed 27 test cases to achieve 100% code coverage using eight parameters: Table 6-V. We analyzed the functionality of the product code and reduced the number of parameters to five and hence the size of the test suite to six using the Each Choice criterion while maintaining the code coverage to 100%. Additionally, we also observed certain missing choices, which were added as a result of the CP technique.

In CS_L1_I, the original test suite contains nine parameters, 560 test cases resulting in 97% code coverage: Table 6-V. After the analysis of the functionality, we suggested the reduction of the input parameters from nine to four, introduced few choices that were missing in the original test suite. The generation of the test cases using the Each Choice criterion resulted in 97% coverage using just six test cases. The 3% code that is never visited belongs to a function, which is never called for execution.

CS_L1_J is just a function with one parameter. The original test suite has one test case which provides 100% code coverage and which is sufficient to test this product code. We however created three choices for this function based on boundary value analysis and created three test cases with CP without loss of coverage.

6.3 Result of implementation of CP on Level 2 case study (CS_L2_A)

Recall that the Level 2 case study is larger in functionality as compared to Level 1 case studies as it makes multiple calls to Level 1 functions in addition to other functionalities. The CP specification for this case study is highly constrained (Table 5-V) so we decided to use three tools for the generation of test frames, i.e., Melba + CASA, ACTS and PICT. Selecting these three tools will help us to observe the impact of test frame generation technique on the cost and quality of test suites. We used PICT for this specific case study because of its high constraint complexity and for that fact that it uses test-based strategy with greedy algorithms as compared to parameters based strategy used by ACTS. The CP specification has 11 parameters and 27 constraints to ensure that the

Table 6-VI: Result of CP implementation for CS_L2_A using CASA, ACTS and PICT

Tools	**Selection criterion**	**Non constrained (-C)**		**Constrained (+C)**		**Coverage (%)**
		Cost	Time (sec)	Cost	Time (sec)	
	Initial test suite	225	NA	NA	NA	86
CASA	**Each Choice**	8	1.013	8	3.13	100
	Pairwise	48	6.47	50	9.28	100
	Three-way	192	3485.57	181	4312.88	100
ACTS	**Each Choice**	8	0.181	8	0.549	100
	Pairwise	50	0.186	56	1.42	100
	Three-way	204	0.195	231	6.42	100
PICT	**Each Choice**	8	0.054	9	0.554	100
	Pairwise	48	0.060	54	0.554	100
	Three-way	235	0.115	242	0.605	100

combination of choices lead to valid test frames. The original test suite for this case study contains 225 test cases with 86% code (statement/line) coverage: Table 6-VI. While analyzing the existing test suite we observed that certain choice combinations were missing and some combinations have more than one occurrence resulting is retesting of the same functionality and hence covering the same part of the code. We also discovered some invalid tests in the original test suite. It was however possible to find the test inputs for those invalid tests since they belonged to priority class one (discussed in section 5.5.1), but they were not contributing to testing the code and in turn were only adding to the cost of the test suite. We were able to avoid such kind of tests by introducing constraints among the choices.

Implementing CP resulted in 100% coverage even for Each Choice, which shows the importance of following this approach. However, the code coverage criterion is statement coverage and it is not very expensive to achieve 100% coverage using this criterion. However, the other aspect of quality, i.e., cost, has seen drastic improvements. We observed that Each Choice, which is the cheapest of all the criteria shows 96% reduction in cost of the test suite and Pairwise is a 76% reduction in cost (CASA) from the initial test suite.

Similarly to few other industrial case studies we generated test frames in both the presence and absence of constraints. Since the initial test suite does not employ constraints, a reason there are some invalid test cases, generation of test frames in the absence of constraints and using CP is a good basis of comparison with the original suite. We observe from Table 6-VI that each application of CP using the three tools, although not accounting for constraints, and therefore including illegal combinations of input

values, achieves 100% statement overage (better than the initial test suite), for a much smaller cost (i.e., number of test cases): cost reduction ranging between 78% and 96%. Further, in the presence of constraints one can observe that, applying CP leads to significant savings when compared to the original test suite (cost reduction ranging between 76% and 96%) while achieving higher structural (statement) coverage (100%).

Since CS_L2_A has 27 constraints, we observed that introducing constraints has led to an increase in the cost of the test suite. We have the same result for all three tools and for Pairwise and three-way selection criteria except for CASA three-way where the cost has dropped compared to when the constraints were not used, although not significantly. We also observed that CASA is the cheapest of the three tools and the number of test frames generated using CASA for Pairwise and three-way is less than the ones generated by ACTS and PICT, concluding that algorithms based on simulated annealing are less expensive than greedy algorithms. However, this comes with a cost; we observed that CASA took the highest time, among the three tools, for the generation of test frames, particularly for the three way criterion with constraints, CASA took approximately 72 minutes whereas ACTS just took 6 sec and PICT 0.6 sec, for generating approximately 24% smaller test suites. Comparing ACTS and PICT we observed that ACTS, which is a parameter based greedy algorithm takes more time than PICT which is a test based algorithm and generates a comparatively smaller test suite in case of the three-way criterion and PICT takes less time than ACTS for the Pairwise criterion and still generates smaller test suites.

6.4 Conclusion

Table 6-VII concludes the results for all the industrial case studies. The original overall code coverage (column 2) is 92.4% and the final overall code coverage using CP technique is 95.6 % (column 3) which is an improvement of 3.2%. This code coverage is obtained at a cost that is 91.6% lower than the original cost: 966 initial test cases are reduced to 81 using the CP technique. (The initial test suite had a pool of 560 tests, which were catering to three product files. We have added that number once, in order to calculate the cost of the original test suites therefore the initial cost is 966.) Among the different selection criteria chosen for the generation of test frames, the ones in the table are those which led to maximum coverage with minimum test cases.

One advantage of CP is that it helps us keep track of parameters that are important for testing the functionalities in product files. Therefore, we kept a close look at the parameters and the values being used in the original test suite. Initially we observed that

Table 6-VII: Result of CP implementation for industrial case studies

Industrial Case Study Product files	**Initial Code Coverage %**	**Final Code coverage %**	**Initial cost**	**Minimum cost to achieve maximum coverage using CP**	**% age change in cost**	**Initial no of parameters**	**No of parameters using CP**	**% age change in no of parameters**
Level 1 Product file 1	80	**80**	1	**2**	**100 ↑**	2	**2**	**0**
Level 1 Product file 2	85	**100 ↑**	584	**26**	**95.5 ↓**	9+4	**7**	**46 ↓**
Level 1 Product file 3	83	**82 ↓**	76	**11**	**85.5 ↓**	11+37	**5+46**	**6 ↑**
Level 1 Product file 4	99	**99**	587	**12**	**97.9 ↓**	9+8	**4+5**	**47 ↓**
Level 1 Product file 5	100	**100**	1	**3**	**200↑**	1	**1**	**0**
Level 1 Product file 6	100	**100**	580	**6**	**98.9 ↓**	9	**6**	**33↓**
Level 1 Product file 7	100	**100**	31	**11**	**64.5 ↓**	8 +7+2	**5 +5+3**	**23.5↓**
Level 1 Product file 8	100	**100**	1	**2**	**100 ↑**	2	**3**	**50 ↑**
Level 2 Product file 9	85	**100 ↑**	225	**8**	**96.4↓**	15	**11**	**26.6**
Total	92.4	**95.6 ↑**	966	**81**	**91.6↓**	124	**103**	**16.9 ↓**

there was an overuse of parameters which were actually not contributing to the testing the product code. Additionally, these parameters lacked values that were important for testing the code. During the process of generating a CP specification, we reduced the number of parameters and introduced all possible choices which were important for testing the respective functionalities. This led to reducing the number of parameters by 16.9%. Reducing the number of parameters means fewer combinations in an n-way adequate test suite, generating fewer test frames and hence smaller cost. Therefore, the CP technique has led to the generation of test suites that improved the code coverage by 3.2% with a cost which is 91.6% smaller than the cost of the original test suite and with 16.9% fewer parameters.

Another important conclusion that we drew from this case study is the impact of the test frame generation techniques (simulated annealing and greedy algorithm) and constraint representation technique on the quality of the test suite. We observed that CASA being a metaheuristic technique generates the same number of or fewer test frames than ACTS and PICT, which are based on greedy algorithms. However, this came with significantly more time taken by CASA to generate the test frames than by ACTS and PICT. This was especially evident during the generation of test frames for highly constrained CP specifications. While comparing our results with the initial test suite we observed that the initial test suite lacked the use of any systematic input modelling technique, nor were they using any automated test frame generation technique thereby resulting in missing or redundant test cases.

Since four of the eleven case studies have constraints among the choice, we have few observations on the impact of constraints on the generation of test frames. We

observed that, for highly constrained CP specifications, such as CS_L2_A, which has constraints of up to 6 tuples, the generation of test frames with constraints takes longer than the generation of test frames without accounting for constraints and we observed this result with all three tools (CASA, ACTS and PICT). For the same case study, we also observed that the test suite generated while accounting for constraints (6-way) is mostly larger, for all three tools, than the one without constraints in order to accommodate complex constraints among the choices. Our observation is in agreement with the observation of Cohen et al. [15]. For the other case studies, which did not have very complex constraints (i.e., CS_L1_A, CS_L1_B and CS_L1_C), we observed that implementing constraints among choices resulted in higher cost for Each Choice but reduced cost for Pairwise. Hence, we can conclude that, if the system under test is highly constrained, introducing constraints will lead to an increase in cost of the test suite in order to accommodate the constraints and vice versa. We studied this phenomenon further in Chapter 7.

We have some conclusions about the CP implementation on the case studies related to LTE middleware. We observe that such case studies have two main characteristics and we have recommendations for such software systems

1. Most of the parameters and therefore categories have choices that represent unique input values, i.e., one choice per discrete value. Since the test inputs will largely remain the same, the quality of the test suite will greatly depend on the test frame generation technique rather than the test input selection technique.

2. Some of the case studies have highly constrained choices usually resulting from the complexity of the domain. Thereby requiring an appropriate tool capable of handling the constraints while generating test frames.

Therefore, it becomes extremely important to select an appropriate algorithm or tool for the generation of test frames that can handle complex constraints and can generate test frames efficiently. Our survey [16] can help a test engineer in making that selection.

Chapter 7 —An Experimental Evaluation of the Impact of Different Types of CP Constraints on the Quality of Test Suites

Regardless of the importance of error/single annotations and constraints among the choices to forbid infeasible combinations, we could not find any study, to the best of our effort, which shows the impact in terms of fault detection effectiveness, code coverage or cost of using (or not) the error/single annotations and constraints among choices. Such a study is the need of the hour because constraint satisfiability is a resource intensive process. If the resources are limited it will not always be possible to implement all kinds of constrains in order to generate feasible test frames. This study will help a tester comprehend a simple question: 'how much of the constraints are enough?' And it will help him/her decide how many constraints (error or single or constraints among choices) will be sufficient for testing the system with given resources. This, hence, became our motivation for the current work.

For conducting these experiments, we designed CP specifications for two academic and three industrial case studies, using various combinations of constraints, i.e., error, single and constraints among choices. These CP specifications are then used to generate test frames using the Each Choice and Pairwise selection criteria with the help of Melba [10, 11] and its integration with CASA [20], ACTS [19], and PICT [21]. The test frames are strategically provided test inputs in order to generate test cases as discussed in section 5.5. We then studied the quality of the test suites as relative cost of the resulting test

suites (i.e., number of test frames), their effectiveness at finding faults (using mutants) and at covering the code.

We formulated different research questions (section 7.1) and designed experiments to answer those research questions (section 7.2). Section 7.3 presents results. Section 7.4 provides a general discussion of the results and we conclude in section 7.5.

7.1 Research Questions

In order to analyze the impact of different kinds of constraints on the quality of the test frames, i.e., fault detection effectiveness, code coverage and test suite cost, we started by identifying the following research questions.

RQ1. What is the impact of annotating choices as Error in the CP specifications?

RQ2. What is the impact of annotating choices as Single in the CP specifications?

RQ3. What is the impact of constraining choices in the CP specification to prevent invalid combinations?

RQ4. What kind of constraint (Single, Error or Constraint among choices) has greatest impact on test suite quality?

7.2 Design of Experiments

Out of all the case studies discussed in Chapter 5, we used two academic case studies, Triangle (section 5.1.2) and NextDate (section 5.1.1), and three industry case studies, CS_L1_A, CS_L1_C and CS_L2_A (section 5.2), for this study. Table 7-I summarizes the CP specification of both types of case studies. Please refer to sections 5.3.1, 5.3.2.1, 5.3.2.3 and 5.3.3 for the description of the table. We did not use the PackHexChar case study because its CP specification does not have choices annotated as Single so we chose the other two academic case studies to observe a clear impact of three

different types of constraints. We chose three industrial case studies out of the total of 13 because of the complexity of constraints in their CP specification. The CP specifications of these case studies have constraints among choices and Single annotations; recall that industrial case studies do not have Error annotations in their CP specification. We however included CS_L2_A in our studies even though its CP specification does not have Error and Single annotations because it is an industrial case study with complex constraints and we mainly wanted to see the behaviour of combinatorial testing tools in the presence of complex constraints. Therefore, all these case studies are more suitable for observing the impact of constraints and Single annotations as compared to the others. We generated different versions of CP specifications in the presence of one or more than one constraints and generated test frames using the integration of CASA and Melba for the academic case studies and using CASA + Melba, ACTS and PICT for the industrial case study. For Industrial case studies (CS_L1_A and CS_L1_C), we used both ACTS and CASA to generate two test suites for comparison and for CS_L2_A we used ACTS,

Table 7-I: CP Specification for the case studies

							Choices involved in X																	
Case study	LOC	Parameters	Categories	Choices	Single Choices	Error Choices	X = all constraints	X = 1 constraint	X = 2 constraints	X = 3 constraints	X = 4 constraints	X = 5 constraints	X = 6 constraints	X = 7 constraints	X = 8 constraints	X = 11 constraints	No. of terms in CNF	2-tuple constraints	3-tuple constraints	4-tuple constraints	5-tuple constraints	6-tuple constraints	%age of choices constrained	Constraint per choice
Triangle	38	3	9	18	3	3	9	3	0	0	6	0	0	0	0	0	12	9	3	0	0	0	66	1.5
NextDate	179	3	4	18	1	6	6	4	1	1	0	0	0	0	0	0	4	3	1	0	0	0	72	0.8
CS_L1_A	65	5	5	16	1	0	11	7	1	2	1	0	0	0	0	0	11	10	1	0	0	0	75	1
CS_L1_C	329	6	6	21	1	0	11	0	6	1	1	0	1	1	0	1	12	2	2	6	2	0	52	1.18
CS_L2_A	1060	11	15	46	0	0	39	21	8	5	0	3	2	0	0	0	27	13	5	7	0	2	85	0.7

CASA and PICT. Since we could not perform fault detection effectiveness for industrial case studies, we used two or three tools to draw assertive conclusions.

To generate test frames in the presence of single and error constraints we used the procedure we described earlier (section 5.4). We then construct test frames using the Each Choice and Pairwise selection criteria and select test inputs for test cases using the systematic procedure mentioned in section 5.5. We then measured the quality of test suites using the measurements that we defined in section 5.6.

In the experiments, even in the presence of constraints when the could-be error choices are not annotated as such and are part of the choice pool, we did not employ any additional constraints for those error choices but rather followed the same priority model described for the absence of constraint (section 5.5.1). The error choices in the CP specifications are not constrained with other choices because they follow the single choice selection criterion where the test inputs for those test frames are manually selected at the end on the basis of permissible choice combinations mainly for robustness testing.

7.2.1 Experimental Setup

The driving force behind designing different experiments is to find an answer to our research questions (section 7.1) and hence conclude on "How many of the constraints are enough?" The experiments are broadly divided into two groups (Table 7-II). In the first group (Experiment 1 to Experiment 6), test frames are generated using the Each Choice and Pairwise criteria while accounting for constraints among the choices in the presence/absence of single or error annotations. In the second group (Experiment 7 to Experiment 12), test frames are generated using the same criteria while omitting constraints among choices but in the presence/absence of single/error annotations.

Table 7-II: Experiments for different constraint types

Experiments	Constraint among choices	Single	Error	Description
Exp 1	+C	-S	-E	• Results in feasible combinations of choices but could-be error/single choices will appear more than once in test frames • Results in more test frames for robustness testing compared to Experiment 6.
Exp 2	+C	-S	!E	• Could-be single choices will appear more than once in test suite. • Error choices are ignored to emphasize testing for acceptable values resulting in no robustness testing. • Useful in comparing results between completely ignoring the error choices and covering them once (Experiment 3).
Exp 3	+C	-S	+E	• Could-be single choices will appear more than once. • Error choices restricted to one occurrence. • Results can be compared with Experiment 1 to single out the impact of annotating error choices.
Exp 4	+C	+S	-E	• Each Single choice appears only once in test frames. • Could-be error choices appear more than once in the test suite. • Comparing results with Experiment 1 will show the impact of single annotations.
Exp 5	+C	+S	!E	• Each Single choice appears only once and error choices are omitted. • Checks the impact of single annotations while completely ignoring error choices (compared to Experiment 4).
Exp 6	+C	+S	+E	• This is the default use of CP. • This is a basis of comparison for all the other experiments.
Exp 7	-C	-S	-E	• The CP specification is devoid of any kind of constraints. • Many infeasible test frames can be expected • This is an unrealistic application of CP but a good basis of comparison with other experiments. For instance, when comparing with Experiment 1 we can study the impact of constraints among choices when other factors are kept constant.
Exp 8	-C	-S	!E	• Error choices are omitted from the CP specification. • Could-be single choices appear multiple times.
Exp 9	-C	-S	+E	• Could-be single choices appear more than once but the error choices appear only once in the set of test frames. • This can be compared to Experiment 7 to see the impact of –E versus +E.
Exp 10	-C	+S	-E	• Single choices appear once while error choices can appear more than once. • This is useful to observe the impact of single annotations while ignoring all other kinds of constraints (comparison with experiment 7).
Exp 11	-C	+S	!E	• Single choices appear once • Error choices are omitted from the CP specification.
Exp 12	-C	+S	+E	• Single and error choices appear once in the absence of constraints among choices. • This can be compared to Experiment 7 to see the impact of introducing single and error annotations and to Experiment 6 to see the impact of +C/-C when other constraints are present.

For each experiment, we alter the original CP specification according to the experiment description. This involves either removing constraints among choices,

removing certain choices or removing the annotations of certain choices so that they are treated as just another choice in the choice pool. Then, test suites are generated for the selected selection criteria, oracles are created, tests are executed against the mutants and the un-mutated software, and test suite quality is measured.

We conducted 12 experiments as described in Table 7-II and assigned a nomenclature for better understanding. Specifically, +C indicates that constraints among choices are part of the CP specification whereas –C indicates that those constraints are omitted from the original CP specification. +S specifies that single annotations are applied on choices. –S specifies that single annotations are omitted and, as a result, the 'could-be' single choice will appear multiple times, i.e., in more than one test frame, combined with more than one permitted choices, depending on the selection criterion, constraints and the test frame generation technique. Similarly, +E specifies that error annotations are used on choices restricting their appearance to one whereas –E specifies that error annotations are omitted resulting in more than one test frame containing the corresponding choice. Experiments with '!E' are designed to handle situations where the test engineers are not interested in robustness testing with out-of-bound values. In these experiments, the error choices are completely ignored and are not even a part of the CP specifications. The next section discusses these experiments in detail.

We illustrate a few of these experiments using a partial CP specification of the NextDate case study in Table 7-III. The complete CP specification is available in appendix B.1 In this table, some choices are not constrained, few are tagged as single or error, while the last three choices have constraints involving other choices. This is a CP specification for Experiment 6 (+C, +S, +E). A test suite formed for this specification

Table 7-III: Example CP Specification

Categories	Choices	Constraints among choices /Single/Error
2:Year	C2.2: CommonYear	
3:Month	C3.1: 30daysMonth	
	C3.3 Month==2	
4:Day	C4.1:Day≥1 && Day<28	Single
	C4.2:Day<1	Error
	C4.3:Day >31	Error
	C4.4:Day==29	(!C4.4 || !C2.2 || !C3.3)
	C4.5: Day==30	(!C4.5 || !C3.3)
	C4.6:Day==31	(!C4.6 || !C3.1) && (!C4.6 || !C3.3)

will have exactly one test frame containing choices C4.1, C4.2 and C4.3 because of Single and Error annotations and the other test frames will honor the constraints. Experiment 5 (+C, +S, !E) will modify this CP specification and will remove C4.2 and C4.3 from the CP specification; As a result no test frame will contain these choices. Further, Experiment 3 (+C, -S, +E) will omit the Single annotation for C4.1 as a result of which C4.1 can appear more than once in the test suite. Experiment 6 is the default use of the CP technique (we use the original CP specification). The other five experiments are degraded versions of Experiment 6, for instance: Experiment 1 has only constraints among choices but the CP specification does not contain any error or single annotation; Experiment 2 is similar to Experiment 1 except that the CP specification does not contain out-of-bound choices (those are specified as error choices in the original specification).

The intent of designing all the experiments using different combinations of single, error and constraints among choices is to observe which type of constraint has a greater effect on the quality of the test frames and hence test cases. For example, in case of the Experiment 7 (–C, -S, -E) and Experiment 9 (–C, -S, +E), it is possible to observe the impact of annotating error choices when the other two factors (C and S) are kept constant. In the next section, we discuss results from these experiments.

7.3 Result of experiments

We first discuss experimental results for each case study separately: Triangle (section 7.3.1), NextDate (section 7.3.2), CS_L1_A (section 7.3.3), CS_L1_C (section 7.3.4) and CS_L2_A (section 7.3.5). We answer the research question in the following section (section 7.4). Each figure in this section shows, for each experiment and the Each Choice and Pairwise selection criteria, the amount of statement (Stmt), branch (Br) and term (Tm) coverage (%), cost (i.e., the total number of test frames), effective frames (the number of test frames that actually kill at least one mutant), and mutation score (%) for the academic case study. For industrial case studies, we report on the use of CASA and ACTS for the Each Choice (EC) and Pairwise (PW) criteria with cost, i.e., number of test frames (TF) and statement coverage (%).

7.3.1 Triangle case study

The results for Triangle for the Each Choice and Pairwise criteria are shown in Figure 7-1 and Figure 7-2 respectively.

7.3.1.1 Structural coverage

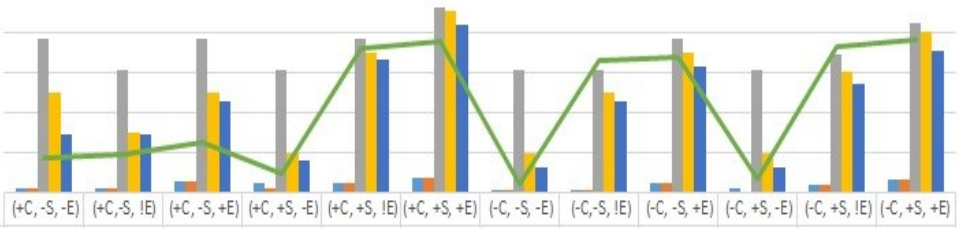
Regardless of the criterion, accounting for both Error and Single choices improves structural coverage: e.g., from (+C, -S, -E) to (+C, +S, +E). This is due to Error choices more so than to Single choices: Error pays more than Single. This is especially clear for Pairwise where term coverage improves by 30% in the absence of single annotation and by 70% in the presence of single annotation when error annotation was introduced in both cases and branch coverage showed an improvement of 20% and 70%. We also observed similar results in the absence of constraints (-C). The annotation of singles in the presence and absence of constraints, when error choices are not annotated results in a

lower or similar code coverage, i.e., (±C, -S, -E to ±C, +S, -E), but never in an improvement. Using choices annotated as single only once in the test suite while not annotating Error choices results in more occurrence of the latter choices, leading to more robustness testing but reduced testing of the functional code (fewer Single choices in test frames). This also applies to results obtained with Each Choice. Ignoring the error choices completely (!E) gave better coverage than when they are in the CP specification but are not annotated as error (-E) but shows reduced coverage than when they are annotated (+E), with both criteria, mainly in the presence of single annotations. !E gives better coverage than –E because when error choices are not a part of the CP specifications, there are no test frames for out-of-bound testing hence the possibility of feasible tests increase. This seems counter intuitive: no robustness testing (!E) leads to more coverage (compared to –E). However, we need to consider constraints. Not annotating error choices as such leads to many choice combinations with the error choice that are not feasible, thereby reducing coverage. This negative impact on coverage is somewhat limited when constraints among choices are used (+C) but increased when those constraints are not used (-C). +E gives better coverage than !E because with +E the test suite has one test frame for each error choice thereby covering code devoted to defensive programming, which is not the case with !E (those choices are omitted).

These observations also apply to experiments where constraints among choices are not accounted for (-C).

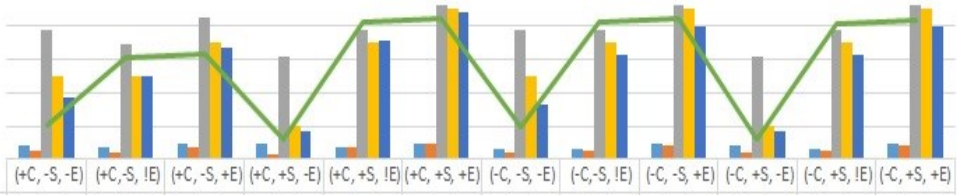
7.3.1.2 Fault detection effectiveness

Mutation score improves as code coverage increases and reaches its maximum when all types of constraints are taken into consideration (+C, +S, +E), regardless of the



	(+C, -S, -E)	(+C,-S, !E)	(+C, -S, +E)	(+C, +S, -E)	(+C, +S, !E)	(+C, +S, +E)	(-C, -S, -E)	(-C,-S, !E)	(-C, -S, +E)	(-C, +S, -E)	(-C, +S, !E)	(-C, +S, +E)
Total frames	3	3	6	5	5	8	2	2	5	3	4	7
Effective Frames	3	3	6	3	5	8	2	2	5	1	4	7
Stmt coverage (%)	76.9	61.5	76.9	61.5	76.9	92.3	61.5	61.5	76.9	61.5	69.2	84.6
Br Coverage (%)	50	30	50	20	70	90	20	50	70	20	60	80
Tm coverage (%)	29.2	29.2	45.8	16.7	66.7	83.3	12.5	45.8	62.5	12.5	54.2	70.8
Mut. Score (%)	18	19	25	10	72	75	5	66	68	7	73	76

Figure 7-1: Results for TRIANGLE: Each Choice Criterion



	(+C, -S, -E)	(+C,-S, !E)	(+C, -S, +E)	(+C, +S, -E)	(+C, +S, !E)	(+C, +S, +E)	(-C, -S, -E)	(-C,-S, !E)	(-C, -S, +E)	(-C, +S, -E)	(-C, +S, !E)	(-C, +S, +E)
Number of frames	8	7	10	9	7	10	6	6	9	8	6	9
Effective Frames	5	4	7	3	7	10	4	5	8	4	5	8
Statement coverage	76.9	69.2	84.6	61.5	76.9	92.3	76.9	76.9	92.3	61.5	76.9	92.3
Branch Coverage	50	50	70	20	70	90	50	70	90	20	70	90
Term coverage	37.5	50	66.7	16.7	70.8	87.5	33.3	62.5	79.2	16.7	62.5	79.2
Mutation Score	21	61	63	12	82	84	20	82	84	12	81	83

Figure 7-2: Results for TRIANGLE: Pairwise Criterion

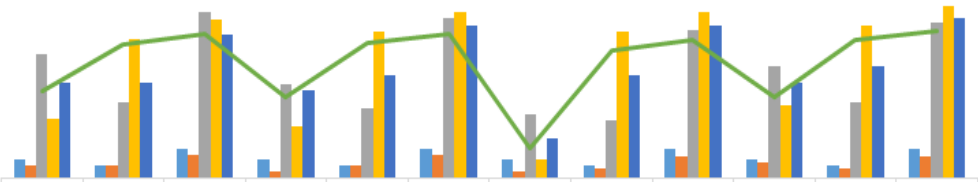
criterion (75% and 84% resp. for Each Choice and Pairwise). When constraints among choices are not used, the mutation score is still maximum when both Single and Error choices are used (-C, +S, +E), regardless of the criterion (76% and 83% resp. for Each Choice and Pairwise). We observed that the test suite with (-C, +S, +E) killed two more mutants than the test suite (+C, +S, +E), this is because the former test suite has test frame for isosceles triangle and with the specific test inputs the impact of which we explained in section 5.6.2, this suite killed more mutants. Making a different test input selection varies the result of the mutation score whereas the structural coverage still

remains the same. So, this difference in mutation score is largely because of the test input selection.When no constraint is used (-C, -S, -E) mutation score is at its worse.

Besides a reduction in mutation score when Single and Error annotations are not used, we also observe a drop in mutation score when only Single annotations are considered without annotating Error, e.g., (±C, -S, -E) to (±C, +S, -E), with both selection criteria. Annotating Single in the absence of error is not very effective whereas the reverse is not true, i.e., annotating Error in the absence of Single greatly improves mutation score.

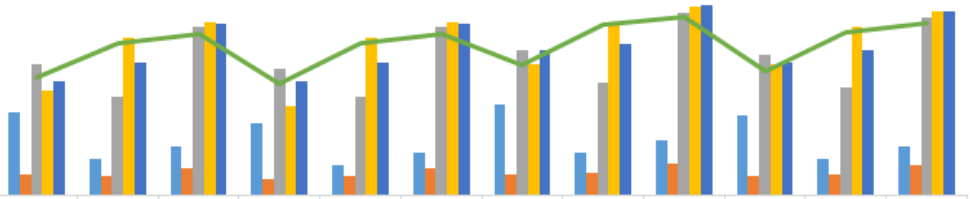
7.3.1.3 Test suite cost

We observe that using Error or Single annotations do not always lead to a reduced number of test frames, on the contrary. When the Error or Single annotations are used, more test frames are needed to form an Each Choice or Pairwise adequate test suite since having Error/Single choices appearing only once reduces chances to form test frames. For similar reasons, complex constraints among choices (which is the case of Triangle, recall section 5.1.2), limiting possibilities for combinations, increase cost. We also observed that as the number of test frames in the presence of constraints increases, effectiveness also increases, which means all test frames were equally contributing to the killing of more mutants. This behaviour is more prominent with Pairwise, e.g., the effectiveness improved from 62% (+C, -S, -E) to 100% (+C, +S, +E) where the effective frames increased from 5 to 10 and 66% (-C, -S, -E) to 89% (-C, +S, +E) where the effective frames increased from 4 to 8.



	(+C, -S, -E)	(+C,-S, !E)	(+C, -S, +E)	(+C, +S, -E)	(+C, +S, !E)	(+C, +S, +E)	(-C, -S, -E)	(-C,-S, !E)	(-C, -S, +E)	(-C, +S, -E)	(-C, +S, !E)	(-C, +S, +E)
Total frames	7	5	11	7	5	11	7	5	11	7	5	11
Effective Frames	5	5	9	3	5	9	3	4	8	6	4	8
Stmt coverage	45.7	28.3	60.9	34.8	26.1	58.7	23.9	21.7	54.3	41.3	28.3	57.1
Br Coverage	22	51.2	58.5	19.5	53.7	61	7.3	53.7	61	26.8	56.1	63.4
Tm coverage	35.3	35.3	52.9	32.4	38.2	55.9	14.7	38.2	55.9	35.3	41.2	58.8
Mut. Score	32	49	53	30	50	53	11	47	51	30	51	54

Figure 7-3: Results for NEXT DATE: Each Choice Criterion



	(+C, -S, -E)	(+C,-S, !E)	(+C, -S, +E)	(+C, +S, -E)	(+C, +S, !E)	(+C, +S, +E)	(-C, -S, -E)	(-C,-S, !E)	(-C, -S, +E)	(-C, +S, -E)	(-C, +S, !E)	(-C, +S, +E)
Total frames	39	17	23	34	14	20	42	20	26	37	17	23
EffectiveFrames	10	9	13	8	9	13	10	11	15	9	10	14
Stmt coverage	60.9	45.7	78.3	58.7	45.7	78.3	67.4	52.2	84.8	65.2	50	82.6
Br Coverage	48.8	73.2	80.5	41.5	73.2	80.5	61	80.5	87.8	61	78	85.4
Tm coverage	52.9	61.8	79.4	52.9	61.8	79.4	67.6	70.6	88.2	61.8	67.6	85.3
Mut. Score	55	71	75	52	71	75	61	79	83	58	76	80

Figure 7-4: Results for NEXT DATE: Pairwise Criterion

7.3.2 NextDate Case Study

Figure 7-3 and Figure 7-4 compile the results for the NextDate case study for Each Choice and Pairwise, respectively.

7.3.2.1 Structural coverage

For the experiments when constraints among the choices are considered, the structural coverage improved from no error and no single annotations (+C, -S, -E) to both error and single annotations (+C, +S, +E) for both selection criteria. We however observed a small decline in coverage in case of (+C, +S, -E) which is akin to what we observed for Triangle: single choices appearing once results in increased occurrences of

error choices which results in a reduced number of feasible test frames. This is also observed in the experiments when no constraints are considered (-C) with Pairwise.

7.3.2.2 Fault Detection effectiveness

Mutation score improves as coverage improves. In the presence or absence of constraints (+C or -C), the mutation score improves when error choices are annotated in the presence or absence of single choices (±S, +E) and reduces when single choices are annotated without annotating error choices (-C, +S, -E) with one exception: a qualitative analysis shows that an Each Choice adequate test suite has a test frame which corresponds to a valid next date and surprisingly does not contain any infeasible test frame, resulting from an error choice. This can be attributed to the non-deterministic behaviour of CASA and perhaps more executions of CASA can change the choice selections.

This case study shows some interesting results. The highest mutation score for Each Choice is for (-C, +S, +E): i.e., error and single annotations are used but constraints among choices are ignored. For Pairwise the highest mutation score is when constraints among choices are ignored and only error annotations are used (83% for –C, -S, +E) which is greater than when all types of constraints are used (75% for +C, +S, +E). The reason is that constraints among choices prevent some choice combinations from occurring in the test frames and these choice combinations, if present in test frames, would perform additional robustness testing than error choices, leading to more coverage and mutation score: an error choice leads to robustness testing involving a single choice, whereas some robustness testing would need to combine several choices. For instance, in (-C, -S, +E) there are test inputs for Feb and day greater than 31 and Sept and day 31

which resulted in killing those mutants; this would not happen with (+C, -S, +E) because constraints are employed to ensure such invalid combinations do not take place which is an outcome of the current definition of Category Partition, which is mainly based on the functionality of the system under test. The definition and implementation of the Category Partition technique can be extended to provide support for robustness.

We also observed that the mutation score of (-C, -S, +E) is greater than that of (-C, +S, +E) in case of Pairwise because of a specific single choice. The choice which is annotated as single is a Day in range 1..27, because we anticipated that it will trigger the same kind of behaviour, irrespective of the month (the next date has the same month as the input date). However there is code in the program which checks the combination of this choice with 30 days month, 31 days month and February. Annotating this choice as single resulted in testing of only one of these three combinations thereby reducing code coverage and mutation score. This observation of ours is inline with the study by Pool et. al. in [14] where they discussed uniformity assumption which is violated in this particular case. In practice this may not however be much of an issue if the Marick's principle [110] is followed whereby black-box tests are first created, for instance with CP, structural coverage of those tests is measured, and additional functional tests are created to increase structural coverage as need be.

7.3.2.3 Test Suite Cost

NextDate does not exhibit as complex constraints (0.84 constraint per choice) as Triangle which explains why Pairwise is not as expensive as Each Choice for NextDate than for Triangle. For experiments with +C, Each Choice results in an increased cost from seven for (+C, -S, -E) to eleven for (+C, +S, +E) while increasing code coverage

and mutation score. In case of Pairwise, cost is reduced to almost 50% when only constraints among the choices are considered (+C, -S, -E) compared to when all types of constraints are considered (+C, +S, +E). The effectiveness of the test suite also improved from 25.6% to 65%. In case of experiments without constraints we observed a similar behaviour for both selection criteria with almost the same results. In case of (+C, -S, +E) to (+C, +S, +E) where the mutation score and code coverage are exactly the same (e.g., mutation score of 75%), we observed that the introduction of single annotations has actually reduced cost (23 to 20) while keeping the number of useful test frames constant (i.e., 13).

7.3.3 Industrial case study CS_L1_A

Figure 7-5 and Figure 7-6 show the results for this case study. Because of the absence of error choices in the CP specification, we have four experiments for this case study. CASA and ACTS are used to obtain test suites and we measure code coverage and cost.

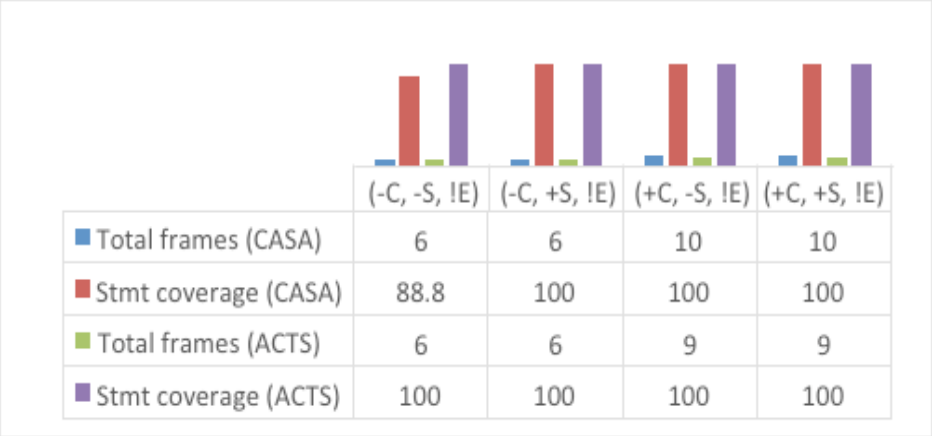
7.3.3.1 Structural coverage

In the absence of constraints (-C), adding single annotations leads to an increased coverage for Each Choice using CASA: The coverage improves from 88.8 % to 100%. ACTS results in 100% code coverage both in the presence and in absence of Single annotations (±S) and constraints among the choices (±C). For the Pairwise criterion, since the test suite has all possible pairs of choices, the structural coverage is always 100%.

7.3.3.2 Test suite cost

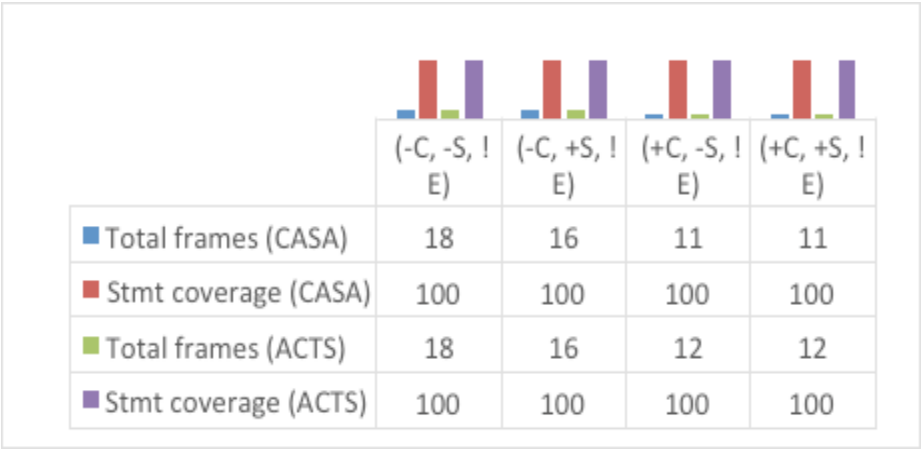
The introduction of single annotations in the presence and in absence of constraints results in the same cost for Each Choice for both tools. For Pairwise, Single annotations in the absence of constraints (-C) result in reduced cost for both CASA and ACTS, whereas, in the presence of constraints (+C), Single annotations result in identical cost but never increased cost.

Introducing constraints among choices (+C) in the presence or in the absence of single annotations (±S) always results in increased cost for Each Choice for both tools.



	(-C, -S, !E)	(-C, +S, !E)	(+C, -S, !E)	(+C, +S, !E)
Total frames (CASA)	6	6	10	10
Stmt coverage (CASA)	88.8	100	100	100
Total frames (ACTS)	6	6	9	9
Stmt coverage (ACTS)	100	100	100	100

Figure 7-5: Results for CS_L1_A Each Choice with CASA and ACTS

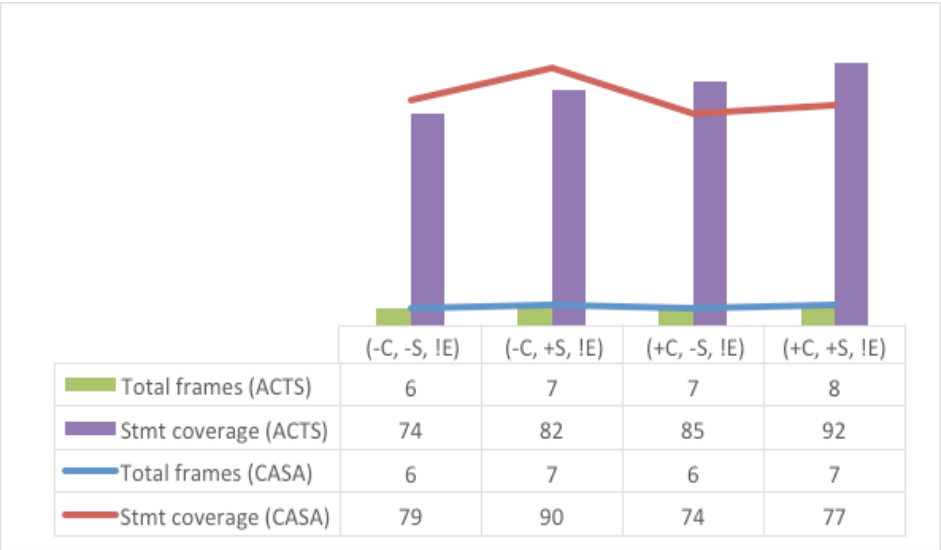


	(-C, -S, ! E)	(-C, +S, ! E)	(+C, -S, ! E)	(+C, +S, ! E)
Total frames (CASA)	18	16	11	11
Stmt coverage (CASA)	100	100	100	100
Total frames (ACTS)	18	16	12	12
Stmt coverage (ACTS)	100	100	100	100

Figure 7-6: Results for CS_L1_A Pairwise with CASA and ACTS

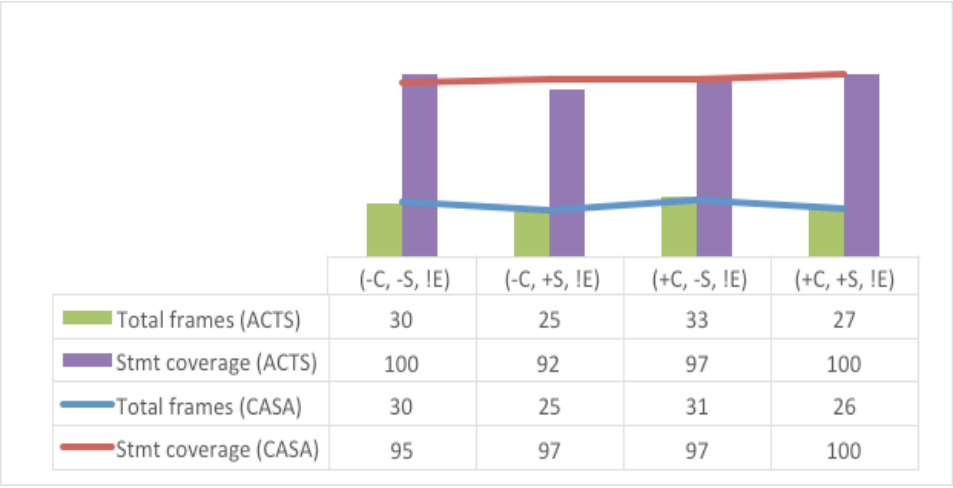
This observation is similar to what we have observed in all the cases studies. In case of Pairwise, introducing constraints (+C) in the presence or absence of single annotations results in reduced cost: It reduces the cost by 38% for CASA and by 33% for ACTS. This is because the constraints in the case study are not as complex as in the Triangle case study. We also observed that (+C) leads to a greater reduction than (+S).

7.3.4 Industrial Case Study CS_L1_C



	(-C, -S, !E)	(-C, +S, !E)	(+C, -S, !E)	(+C, +S, !E)
Total frames (ACTS)	6	7	7	8
Stmt coverage (ACTS)	74	82	85	92
Total frames (CASA)	6	7	6	7
Stmt coverage (CASA)	79	90	74	77

Figure 7-7: Results for CS_L1_C Each Choice with CASA and ACTS



	(-C, -S, !E)	(-C, +S, !E)	(+C, -S, !E)	(+C, +S, !E)
Total frames (ACTS)	30	25	33	27
Stmt coverage (ACTS)	100	92	97	100
Total frames (CASA)	30	25	31	26
Stmt coverage (CASA)	95	97	97	100

Figure 7-8: Results for CS_L1_C Pairwise with CASA and ACTS

Figure 7-7 and Figure 7-8 show the results for this case study. Similarly to the previous case study, we only have four experiments and use both CASA and ACTS.

7.3.4.1 Structural Coverage

In the absence and presence of constraints, i.e., from (-C, -S, !E) to (-C, +S, !E) and from (+C, -S, !E) to (+C, +S, !E), adding single annotations always leads to better statement coverage, a result in agreement with academic case studies when single choices are annotated meticulously. We obtained this result both for CASA and ACTS and for both Each Choice and Pairwise, with one exception. We did a qualitative analysis of that test suite and concluded this was a coincidence resulting from a specific choice combination. In the absence of single annotations (-S), (+C) does not always lead to better coverage, a result also observed from academic case studies. But in the presence of single (+S), (+C) results in better code coverage (with one exception in case of CASA). Again, this result is very similar to that of academic case studies, when singles are annotated meticulously.

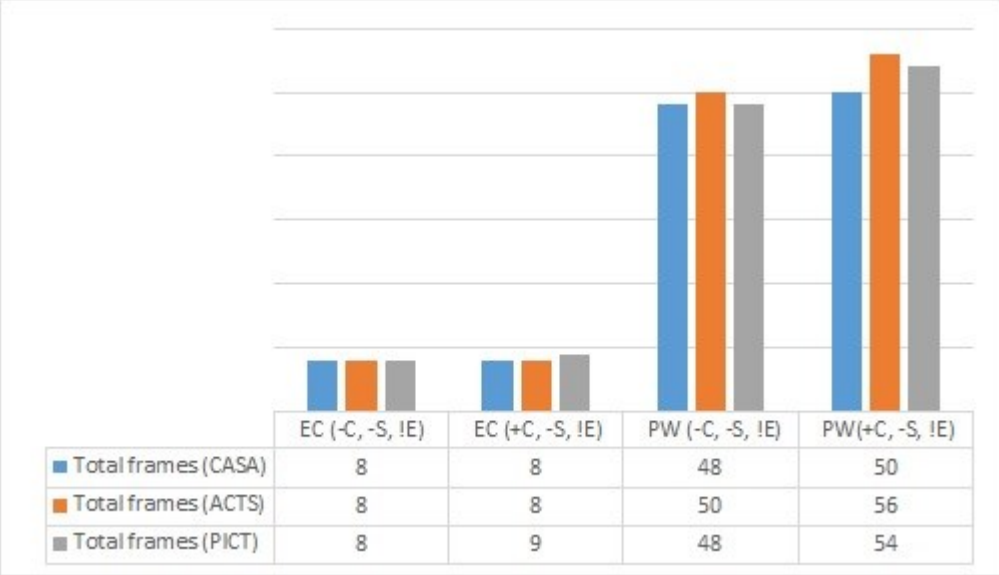
7.3.4.2 Test Suite Cost

Single annotations result in increased cost irrespective of the presence or absence of constraints in case of Each Choice. We observe this result in all experiments with academic case studies. In case of Pairwise, single annotations result in reduced cost. This observation is similar to what we made for NextDate where the number of choices per categories is larger as compared to Triangle, where we do not obtain similar results (for further explanation see section 7.4.2). We also observe that adding constraints (+C) leads to the same or an increased cost but never reduces cost, both in the presence or absence of

single annotations. We have a similar result from Triangle because the constraints among choices in Triangle are complex like this case study and unlike NextDate (section 5.1.1) which has comparatively relaxed constraints. Our observation is also in agreement with the observation by Cohen et. al. [15] where generation of covering arrays in the presence of constraints leads to a larger test suite as compared to the examples where constraints are ignored. Similar to us, they observe this behaviour using more than one covering array generation algorithm.

7.3.5 Industrial Case Study CS_L2_A

Figure 7-9 shows the results of this case study. This is the most complex of all the industrial case studies, having up to 6-tuple constraints. The CP specifications for this cases study does not contain choices annotated as single, therefore we have only two experiments per selection criteria. For this case study, we used three test generation tools: CASA, ACTS and PICT.



	EC (-C, -S, !E)	EC (+C, -S, !E)	PW (-C, -S, !E)	PW(+C, -S, !E)
Total frames (CASA)	8	8	48	50
Total frames (ACTS)	8	8	50	56
Total frames (PICT)	8	9	48	54

Figure 7-9: Cost for CS_L2_A for Each choice, Pairwise using CASA, ACTS and PICT

7.3.5.1 Structural coverage

The implementation of CP resulted in 100% coverage both in the presence and in the absence of constraints, using all three tools and both selection criteria.

7.3.5.2 Test suite cost

The size of the CP specification for this case study is large (11 parameters, 15 categories and 46 choices) and the choices are highly constrained. Having said that, the generation of test frames, in the presence of constraints leads to an increased cost. In case of Each Choice, using CASA and ACTS results in the same cost both in the presence and in the absence of constraints where PICT increased the cost by one test frame. Pairwise on the other hand results in an increased cost because of the introduction of constraints (+C), for all three tools. The time taken by these tools for the generation of test frames also varied the details of which are discussed in section 6.3.

As far as tools for the generation of test frames are concerned, we observe that CASA is the cheapest of all the three tools in terms of the size of the generated test suite which is mainly evident in case of Pairwise criterion and in the presence of constraints. Between the other two tools that are based on greedy algorithm, ACTS is costlier than PICT for Pairwise criterion.

7.4 Discussion on research questions

In this section, coalescing results from the previous sections, we answer the research questions. In order to derive the results we compare experiments where a specific constraint type is not present to where it is present, keeping the other two constraint types unchanged, e.g., in order to see the impact of error annotations we compare (-C, -S, -E)

with (-C, -S, +E) and (+C, +S, -E) with (+C, +S, +E). Figure 7-10, Figure 7-11, Figure 7-12 and Figure 7-13 show the impact of various kinds of constraints on mutation score. We observe similar trends for code coverage (Figure 7-1, Figure 7-2, Figure 7-3, Figure 7-4) and we therefore do not show coverage in Figure 7-10, Figure 7-11, Figure 7-12 and Figure 7-13.

7.4.1 RQ1. What is the impact of annotating choices as Error in the CP specification?

Figure 7-10 shows the impact of error annotations on mutation score for both academic case studies and selection criteria. We are comparing the impact when no other constraint is present and when all other constraints are present in all the four samples (Triangle EC, Triangle PW, NextDate EC, NextDate PW). We observe that when no other constraint is present, having Error annotations (i.e., from –E to +E) improves code coverage (Figure 7-1, Figure 7-2, Figure 7-3, Figure 7-4) and mutation score (Figure 7-10). When all constraints are present, having Error annotations further improves

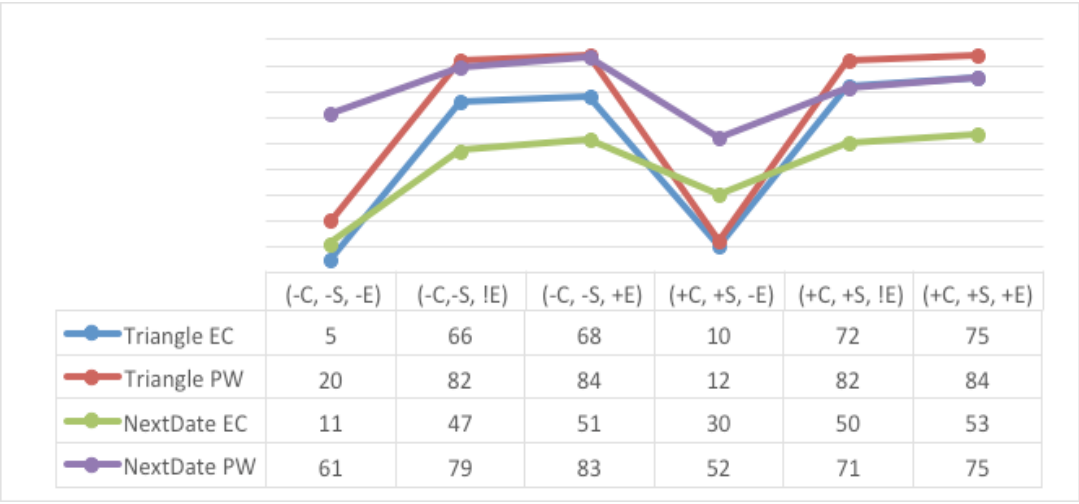


Figure 7-10: Impact of Error annotations on mutation score

mutation score with the exception of the NextDate Pairwise scenario where including constraints led to reduced robustness testing; as a result the mutants created for the related error handling code were not killed (for details check section 7.3.2.2). Further, if the tester decides to completely ignore error choices from the CP specifications (!E), i.e., decides to exclude test cases for the out-of-bound values, this improves coverage and hence mutation score as compared to when error annotation are not used (-E) but results are still a bit lower than when error annotations are used (+E).

We have interesting conclusions about the impact of Error annotations on the cost of the test suite. We comprehended that the cost of the test suite depends on two factors: **Combination Strength** and **Number of Choices per Category**. When Error annotations are introduced with lower combination strength, i.e., t=1, we observe an increase in the cost of the test suite. This is mainly credited to the fact that Each Choice (t=1) is a relatively inexpensive selection criterion and a relatively smaller sized test suite can easily be Each Choice adequate. In the presence of constraints, i.e., +C and +E, more test frames are required in order to honor these constraints and to obtain an adequate test suite. In case of higher strength combinations, the cost of the test suite depends on an additional factor, i.e., the number of choices per category because the more the number of choices per category the more opportunities for choice combinations.

We will illustrate this with the help of a synthetic example (Table 7-IV): Consider a CP specification A with three categories and 12 choices where each category has four choices. If we generate test frames for this specification using ACTS, we will have four test frames for the Each Choice criterion and 16 test frames for the Pairwise criterion. Now consider a CP specification B that has six categories and 12 choices with two

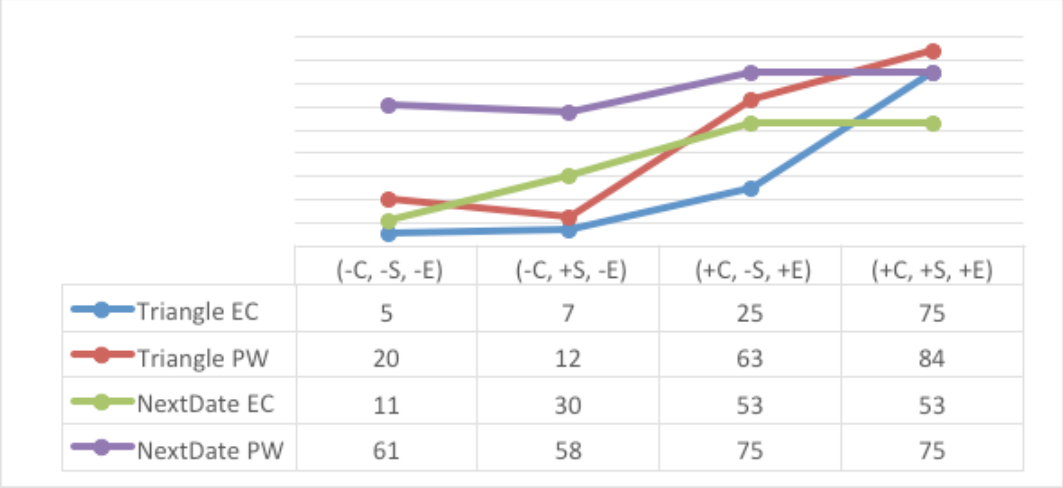
Table 7-IV: Example CP specifications

	Specification A	Specification B
No. of Categories	3	6
Choices per category	4	2
Total Choices	12	12
No. of test frames using EC criterion (t=1)	4	2
No. of test frames using Pairwise criterion (t=2)	16	9

choices per category. The number of test frames generated for this specification is two using the Each Choice criterion and nine using the Pairwise criterion. This illustrates that we get a bigger test suite if there are larger number of choices per category. We have similar observation from our experiments. We observed that when the number of choices per category is small (e.g., Triangle case study), a small number of test frames can create an adequate test suite. However, when Error annotations are introduced, we need more test frames to account for missing combinations for the Each Choice criterion. Even for the Pairwise criterion, when the number of choices per category is small, annotating choices as Error leads to more test frames. On the other hand, for case studies with a large number of choices per category, e.g., NextDate, we observe a comparatively larger number of test frames using the Pairwise selection criterion, and introducing Error annotations does not increase the cost as there are already many test frames to adjust for the missing combinations. We observed this for experiments with constraints and without constraints for both the academic case studies. We didn't report this behaviour for industrial case studies since there are no error choices in their CP specifications.

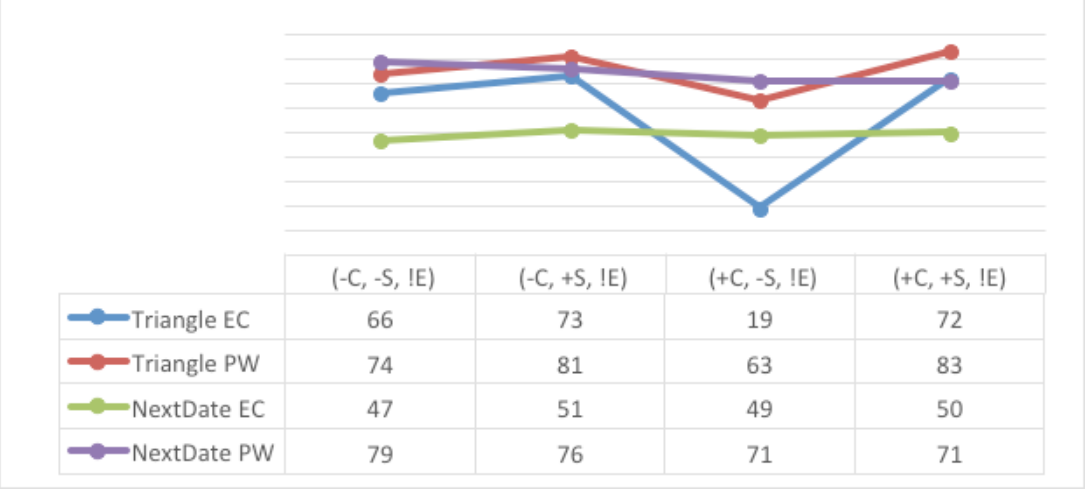
7.4.2 RQ2. What is the impact of annotating choices as Single in the CP specification?

Figure 7-11 shows the impact of Single annotations on mutation score for all four samples of Triangle and NextDate using the Each Choice and Pairwise selection criteria in the presence and absence of other constraints. We observe that Single annotations in the absence of all other kinds of constraints (-C, -E) are not very effective since the could-be error choices and constraints among the choices are not identified; we obtain



	(-C, -S, -E)	(-C, +S, -E)	(+C, -S, +E)	(+C, +S, +E)
Triangle EC	5	7	25	75
Triangle PW	20	12	63	84
NextDate EC	11	30	53	53
NextDate PW	61	58	75	75

Figure 7-11: Impact of Single annotation on the mutation score



	(-C, -S, !E)	(-C, +S, !E)	(+C, -S, !E)	(+C, +S, !E)
Triangle EC	66	73	19	72
Triangle PW	74	81	63	83
NextDate EC	47	51	49	50
NextDate PW	79	76	71	71

Figure 7-12: Impact of single annotations on mutation score (!E)

invalid test frames. However, in the presence of all other kinds of constraints single annotations have a positive or a neutral effect.

Interestingly, we also observe that when error choices are completely ignored (!E) (Figure 7-12), using single annotations in accordance to their purpose (i.e., using them for functionalities which do not need to be tested several times), results in improved mutation score, improved code coverage in lesser or equal cost with higher test suite effectiveness with an exception for the NextDate Pairwise sample (see section 7.3.2.2 for an explanation). We make the same conclusion with CS_L1_A and CD_L1_C, i.e., in the absence of error choices (!E), Single annotations always lead to better coverage with increased or equal cost for Each Choice and reduced cost for Pairwise.

Similar to Error annotations, we observe that the cost of the test suite using Single annotations also depends on two aspects: **Combination Strength** and **Number of Choices per Category**. For the smaller combination strength, i.e., t = 1 (Each Choice criterion), Single annotation results in an increased or same cost since there are smaller number of test frames in a test suite and in order to achieve an adequate test suite more frames needs to be added. Using the Pairwise (t=2) selection criterion for a CP specification when Single annotations are used results in a reduced cost compared to when single annotations are not used. This is where the second factor, the number of choices per category, comes into effect. We observed that when the number of choices per category is small (e.g., Triangle case study), even the Pairwise selection criterion results in increasing the cost, as more combinations are required to form an adequate test suite. In the Triangle case study, there are two choices per category; therefore using

Single annotations with Pairwise is costlier than the other three case studies where the categories have more than two choices.

7.4.2.1 Impact of Single annotations on the quality of a test suite

In this section, we discuss how Single annotations help to improve the quality of a test suite as compared to when single annotations are not used. The purpose of Single annotations is to limit the occurrence of a single-annotated-choice to one in a test suite. If the single annotated choice is not constrained with any other choice, any combination of this choice with choices of other categories can form a test frame. In case the single choice is constrained, the combination should satisfy the constraints. It is however very important that this selection of Single annotations should be done meticulously and the choice chosen as single is true to its purpose. This section illustrates how such an annotation can help improve the quality of the test suite by limiting the single choice occurrence to one and increasing the occurrence of other choices. We illustrate this phenomenon at the level of test frames using an example synthetic CP specification.

Consider a CP specification with three categories A, B and C, where A has three choices (a1, a2, a3) and B and C have two choices each (b1, b2) and (c1, c2) as shown in Table 7-V.

We use the Pairwise and Three-way selection criteria to create sample test frames using ACTS (Table 7-VI and Table 7-VII). We used ACTS because of its deterministic behavior.

Table 7-VI (a) shows six test frames obtained using the Pairwise criterion without annotating the choices as Single. In the second Table 7-VI (b), we annotated choice b2 as single. With this annotation, we are discouraging the combination of b2 with every other

choice, which in turn forces the combinations of b1 (the only other choice in category B) with choices of other categories to form an adequate test suite. A (+S) test suite in Table 7-VI (b), has more occurrences of b1 as compared to a (–S) test suite in Table 7-VI (a). This creates many choice combinations in (+S) which were otherwise not possible in (-S). For example, (a2, b1, c2) is present in (+S) only because b2 has been replaced by b1 (marked as @ in Table 7-VI). Similarly, (a3, b1, c2) is obtained from (a3, b2, c2) because of the Single annotation (marked as # in Table 7-VI) and so is (a1, b1, c1) which is marked as * in Table 7-VI. The Single annotation of b2 has added three test frames (frames 1, 4, 6) to the test suite in Table 7-VI (b) which were otherwise not present in Table 7-VI (a). This process results in more test frames with b1, which is a consequence of reducing the occurrence of test frames containing b2 to one.

Table 7-V: Example CP specification

Category A	Category B	Category C
a1	b1	c1
a2	b2	c2
a3		

Table 7-VI: (a) Pairwise without single annotations (b) Pairwise with single annotations

Pairwise (-S)				
1	a1	b1	c2	
2	a1	b2	c1	*
3	a2	b1	c1	
4	a2	b2	c2	@
5	a3	b1	c1	
6	a3	b2	c2	#

Pairwise (+S [b2])				
1	a1	b1	c1	*
2	a1	b1	c2	
3	a2	b1	c1	
4	a2	b1	c2	@
5	a3	b1	c1	
6	a3	b1	c2	#
7	a1	b2	c1	

Table 7-VII: (a) Three way without single annotations (b) Three way with single annotations

Three way (-S)			
1	a1	b1	c1
2	a1	b1	c2
3	a1	b2	c1
4	a1	b2	c2
5	a2	b1	c1
6	a2	b1	c2
7	a2	b2	c1
8	a2	b2	c2
9	a3	b1	c1
10	a3	b1	c2
11	a3	b2	c1
12	a3	b2	c2

Three way (+S [b2])			
1	a1	b1	c1
2	a1	b1	c2
3	a2	b1	c1
4	a2	b1	c2
5	a3	b1	c1
6	a3	b1	c2
7	a1	b2	c1

It can be argued that an increased number of occurrences of b1 further results in reducing the number of occurrences of b2 which can consequently lead to the reduced coverage for the code that realizes b2. A counter argument is based entirely on the logic of the Single annotation and the assumption of uniformity [14]. A uniformity assumption

ensures that this single annotation will always process the choice uniformly and will have the same impact on the quality of the test suite. The reason of b2's selection as single is based on the fact that more than one occurrence of b2, in a test suite, will result in re-testing of the same functionality and consequently revisiting of the same code: it is not deemed that the additional combinations would bring additional benefits. Therefore, there are fewer chances that the reduced occurrence of b2 would deteriorate the quality of the test suite. On the other hand, it is beneficial to combine b1 with other choices (a reason it is not selected as single) and such a test suite can lead to improved coverage. Further, if there is a concern about which combination with b2 will result in an effective coverage, a constraint among the choices along with error annotation can help to figure that out. All our case studies support this argument. For example, in the Triangle case study we have seen test suite with (+C, +S, +E) performing better than (+C, -S, +E) where the single annotated choices are also constrained.

In Table 7-VI, we also observe that the single annotation results in an increased cost. From our experimental evaluations, we observe that the cost increases if the number of choices per category is small or if the combination strength is small. In case the number of choices per category is large and the strength of the combination is also larger than one, i.e., at least Pairwise, the introduction of single annotations (+S) results in reduced cost: e.g., the industrial case studies and the NextDate case study have larger numbers of choices per category as compared to Triangle and so they have resulted in reduced cost with single annotations. On the other hand, in all our experiments, Single annotations with the Each Choice criterion have resulted in a greater or similar cost.

We further observe that the annotation of choices as Single is only advantageous if the combinatorial strength is less than N where N is the total number of categories in the CP specification. In case the strength of combination is N (i.e., All Combinations), the test suite obtained for the CP specification that has Single annotations does not show any new/different test frames from the test suite obtained from the CP specification in which Single annotations are not used. For example, in Table 7-VII for the Three-way criterion, since the total number of categories is three making the selection criterion All combination, the –S test suite (Table 7-VII (a)) will have all possible combinations. When we annotate b2 as Single and generate test frames using the All Combination (Three-way) criterion we observe that no new test frame is obtained because all the test frames from the +S suite (Table 7-VII (b)) are already present in the –S test suite (Table 7-VII (a)). Therefore, a Single annotation in this case does not introduce any new test frame but certainly results in reducing the cost of the test suite. In this example, we observe a cost reduction of 41.6%.

Similar to the example CP specification discussed above, single annotations also had an impact on our case studies. We observed similar behavior with the Triangle, the NextDate and the industrial case studies. In the Triangle case study while generating test frames for the Pairwise criterion using CASA (and Melba) we observed that annotating choices as single results in reducing the occurrence of single annotated choices and at the same time this increases the combination of other non-single choices resulting in better code coverage and mutation score. In the Triangle case study, choices that check the inequality of the triangle are annotated as single (e.g., a>b+c). It is therefore sufficient to have one test frame with such a choice. When choices are not annotated as single (+C, -S,

+E), this results in more occurrences of test frames for checking the inequality of the sides of the triangle, resulting in more test cases with invalid sides, e.g., test inputs (15, 4, 4) and (16, 5, 6). On the other hand, when choices are marked as single (+C, +S, +E) there would be only one occurrence of test frame to check the triangle inequality. Any additional test frame would correspond to a valid triangle. For example, instead of test input (16, 5, 6) in the (-S) test suite, a test input (7, 5, 6) will be added in the (+S) test suite, which will be a test for a valid scalene triangle rather than an invalid triangle. This would consequently lead to a better test suite quality. We also observed this behaviour for industrial case study CS_L1_C: a single annotation in its CP specification and generation of test frames for the Pairwise criterion using both CASA and ACTS. With these single annotations, we clearly observed better code coverage and a reduced cost.

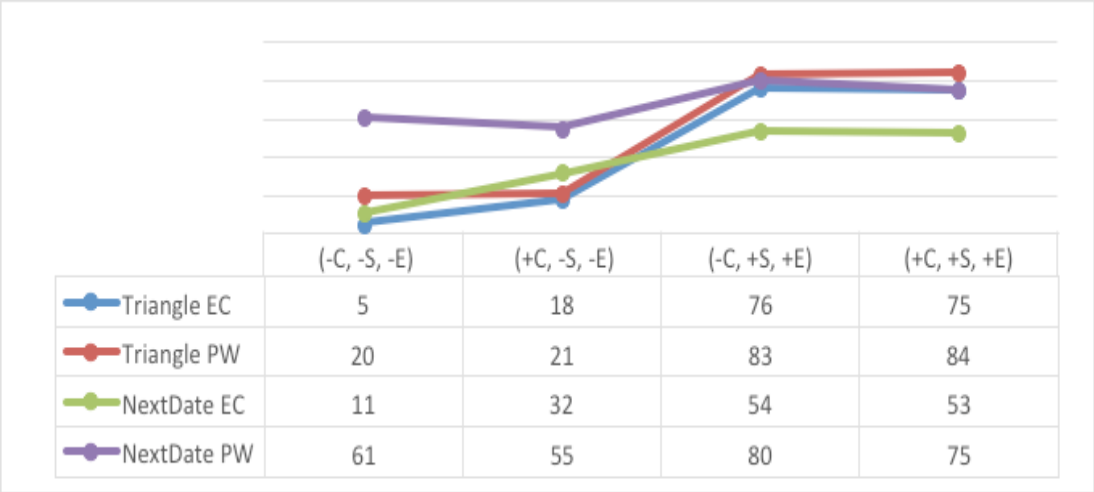
We can summarize our findings as follows:

- The introduction of single annotations improves code coverage and mutation score and therefore the quality of the test suite because of the increased combinations among non-single annotated choices.
- Single annotations always result in increasing the cost for criteria with a low combination strength (t=1, Each Choice).
- For other combination strength (t>1), if there are few choices per category, using single annotations results in increasing the cost e.g., triangle case study. However, for large numbers of choices per category, single annotations reduce cost e.g., CS_L1_C.
- If the strength of combination is equal to the number of categories in the CP specification, then single annotations would only result in reducing the cost of the

test suite (number of test frames) and would not introduce any additional/new test frame.

7.4.3 RQ3. What is the impact of constraining choices in a CP specification to prevent invalid combinations?

Figure 7-13 shows the impact of introducing constraints among choices on mutation score when error and single annotations are not used and when they are. We observe that introducing constraints among choices in the absence of all other annotations improves mutation score for Each Choice but not much for Pairwise. In the case of Pairwise we observe that introducing constraints among choices, i.e., from (-C, -S, -E) to (+C, -S, -E), leads to more occurrences of could-be error and single choices in order to satisfy all the pairs. More occurrences of these choices, particularly error choices, leads to more infeasible combinations hence leading to a drop in quality of test frames. Instead, in case of (+S, +E) there is one occurrence of such error choices hence quality is maintained. Therefore, in the absence of all other annotations, adding constraints among choices does not significantly affect mutation score. We observed with CS_L1_C that using constraints in the presence of Single annotations is more effective than their absence: an observation inline with our academic case studies. We see CASA (Each Choice) in Figure 7-7 performing abruptly whereas ACTS followed the pattern. This can also be due to CASA's non-deterministic behaviour and subsequent executions of CASA can lead to an outcome similar to ACTS. We also observe that for case studies with complex constraints (Triangle, CS_L1_C and CS_L2_A), constraints among choices (+C) result in an increased cost specifically for the Pairwise criterion



	(-C, -S, -E)	(+C, -S, -E)	(-C, +S, +E)	(+C, +S, +E)
Triangle EC	5	18	76	75
Triangle PW	20	21	83	84
NextDate EC	11	32	54	53
NextDate PW	61	55	80	75

Figure 7-13: Impact of constraints among choices on mutation score

We have some interesting observations for the cost of the test suite in the presence and absence of constraints. We observe that the presence of constraints among choices does not always result in reducing the cost of the test suite rather the cost depends on three factors: **combination strength, number of choices per category** and **complexity of constraints in the CP specification.** Similar to the discussions for Error and Single annotations (sections 7.4.1 and 7.4.2), the presence of constraints among choices (+C) in the presence or absence of other constraints (±S, ±E) results in an increased or the same cost for a low combination strength, i.e., Each Choice. For the higher combination strength, i.e., Pairwise criterion, we observe that the cost of the test suite further depends on the **number of choices per category** and the **complexity of constraints** in the CP specification. Recall sections 5.3.1, 5.3.2 and 5.3.3, where we measure the complexity of the CP specification and the constraints employed therein. Based on our experiments we observe that for a CP specification that has complex constraints among choices and few choices per category, criteria result in a costlier test suite. Precisely, we observe that, if there is more than one choice involved in four or more constraints and there is more than one n-tuple constraint (where $n \geq 3$), then there are chances that the number of test frames

generated in the presence of constraints will be larger than those generated in their absence. Using this proposition, we can conclude that the generation of test frames using the Pairwise criterion for five constrained CP specifications (Table 7-I) results in a reduced cost for NextDate and CS_L1_A and an increased cost for Triangle, CS_L1_C and CD_L2_A in the presence of constraints.

7.4.4 RQ4. What kind of constraint (Single, Error or constraint among choices) has greatest impact on test suite quality?

We conclude from the experiments and discussions that among all types of constraints, Error (E), Single (S) and Constraints among choices (C), for the systems with defensive programming, Error annotations (+E) are the most effective at improving test suite quality as compared to a CP specification which has only Single annotations (+S) or only constraints among the choice (+C). For instance, Error annotations solely improved the mutation score from 20 to 84 when no other constraints are present and from 12 to 84 when all other constraints are present for Triangle, using the Pairwise criterion. Talking of pairs, CP specifications containing Error and Single annotations together are more effective (i.e., fault detection, coverage) both in the presence and absence of other constraints (±C). These types of constraints help separate out error choices, which are later added to the test suite using the Single Error Coverage Criterion. Each Single choice is also added in one test suite, which is useful in reducing redundancy. Separating out Error and Single choices, with the help of annotations, leaves more opportunities for combining other choices, which therefore leads to better coverage and effectiveness at finding faults. We observed that moving from (-C, -S, -E) to (-C, +S, +E) improves mutation score from 5% to 76% for Triangle with Each Choice, 20% to 83% for Triangle

with Pairwise, 11% to 54% for NextDate with Each Choice and 61% to 80% for NextDate with Pairwise, making combination (-C, +S, +E) the most effective.

For a system which is built through contracts rather than defensive programming, e.g., our industrial case studies, we observe that if the CP specification has choices which can be annotated as single then such annotations in the presence or absence of constraints (± C) mostly lead to better coverage at a lower cost (for combination strength t>1). On the other hand the addition of constraints among choices in the absence of single annotation (-S), if they are otherwise a part of CP specification, (+C, -S), does not always lead to better coverage, but in the presence of single annotations (+C, +S) certainly improves coverage, although with an additional cost if the choices are highly constrained. Additionally, if there is no single annotation in the CP specification, then constraints among choices are the only option to improve the quality of the test suite and cost depends on the complexity of the constraints.

7.5 Conclusion

Since the technique of applying CP is based on the expertise of the test engineer, one might begin by identifying a lot of categories and choices which when combined to form test frames generate an innumerable number of test frames which can be infeasible if the choices are not competently constrained. The analysis of the quality of the test suite containing infeasible frames, in the present work, shows that they result in lower fault detection, lower code coverage and possibly larger cost. In order to combat this issue Ostrand and Balcer [4] suggest to introduce constraints among choices together with annotating some choices as single or error, since this will ensure that the test frames are

not redundantly testing the same combinations, which are perhaps infeasible, or testing the same functionality or code more than necessary.

The question "How much of these constraints (error, single and constraints among choices) are enough?" greatly remains unanswered in the literature. We tried to answer this question using our experimental evaluation. Further using these experiments, we have also tried to discover and analyze the factors that affect the cost of the test suite when single, error and constraints among the choices are introduced.

Cohen et. al. [15] mentioned that constraints can result in increasing the size of the test suite since more frames are required to accommodate the constraints and generate an adequate test suite. However, there is no mention of how much constraints or what complexity of constraints can affect the size of a test suite. We could not find any supporting literature to the best of our knowledge. Therefore in the present work, we have precisely tried to point out, with the help of experiments, the complexity of constraints

Table 7-VIII: Impact of constraints on Code coverage and Mutation score

Error annotations	**Single annotations**	**Constraints among choices**	**Code coverage**	**Mutation score**
Error impact	-S	-C	↑	↑
Error Impact	+S	+C	↑	↑
-E	Single impact	-C	↑↓	↑↓
!E	Single impact	±C	↑⇔	↑⇔
+E	Single impact	+C	↑	↑
-E	-S	Constraint impact	↑↓	↑↓
!E	-S	Constraint impact	↑↓	↑↓
+E	+S	Constraint impact	↑⇔	↑⇔
!E	+S	Constraint impact	↑⇔	↑⇔

beyond which the cost of the test suite will increase. If the complexity of the constraints is lower than the specified level then the introduction of constraints would lead to a reduction in the size of the test suite. Such a study can help a test engineer evaluate the impact of constraints on the cost of the test suite when a selection criterion is used.

The basis of our evaluation is three industrial and two academic case studies, two selection criteria, two/three technologies for the generation of test frames and twelve experiments. The results obtained from these experiments are mostly consistent and follow almost the same trends for all the case studies and both selection criteria (there are a few exceptions, which are mainly due to the selection of the single choices or the technique for test frame generation.). We compiled the results of the impact of error, single and constraints among choices on the code coverage and mutation score in Table 7-VIII taking into consideration the general trends. Column 1, 2 or 3 in Table 7-VIII shows the impact of a specific constraint when the remaining two constraints are kept constant. The last two columns show the impact of constraint on code coverage and mutation score. An upward arrow (↑) depicts a rise in mutation score and code coverage whereas both upward and downward arrows (↑↓) show mixed results which do not allow us to reach any conclusion. The combination of an upwards and an equal arrow (↑⇔) symbolises a rise or an identical code coverage and mutation score.

We make our conclusions as follows:

- For defensive programming code, if one constraint is to be used, the most effective one is Error annotations because this results in separating out the error (out of bound) choices resulting in a greater chance of having feasible test frames (rows 1, 2 in Table 7-VIII)

- For code with no defensive programming, Single annotations perform well or equal regardless of the presence of constraints among choices, provided single annotations are selected wisely (rows 4 in Table 7-VIII).
- In the presence of could-be error choices that are not annotated as such, single annotations are not as effective because this results in constraining the 'single choice contained' test frame to one occurrence hence increasing the chance of occurrence of could-be error choice in test frames (rows 3 in Table 7-VIII). However if single annotations are used along with error ones and constraints among choices, the results are better than with only error annotations (rows 5 in Table 7-VIII)..
- For defensive programming code, if two types (out of three) of constraints are to be used in a CP specification, Error and Single annotations used together is the most effective selection. The quality of the generated test suite with only Error annotations (-S, +E) is improved with both Error and Single annotation (+S, +E). Needless to say that the single choices should be selected wisely and their selection should be in accordance to their purpose, i.e., it should correspond to a functionality that is required to be tested just once; otherwise, this can result in reducing the quality of the test suite.
- The number of test cases contributing to test suite effectiveness, i.e., the number of frames contributing to the killing of mutants, increases as more constraints are introduced hence reducing the number of useless test frames.

- Specifying complex constraints and a large number of Single or Error choices can result in improving the quality of test suites (rows 8, 9 in Table 7-VIII) and in an increase in the number of test frames (cost).
- The annotation of choices as Single or Error results in increasing the cost when using the Each Choice criterion.
- If the criterion used for the generation of test frames is Pairwise then test frames generated using Single and Error annotations would result in an increased cost if there are few choices per category. If there is a large number of choices per category then Single and Error annotations will reduce the cost.
- In the presence of constraints among choices, the cost increases for Each Choice whereas for the Pairwise criterion the cost of the test suite depends on two more factors: the complexity of constraints in the CP specification and the number of choices per category. If there is more than one choice involved in four or more constraints and there is more than one n-tuple constraint, where $n \geq 3$, then there are chances that the number of test frames generated in the presence of constraints will be larger than those generated in their absence. Further, if the number of choices per category is small, cost is high. If both the factors are positive, i.e., complex constraints and smaller choices per category, the cost for the Pairwise criterion in the presence of constraints is more than in their absence. If either of the factors is negative, the results can vary. This study can help a tester decide the tradeoff between the complexity of constraint and the size of the test suite.
- We also observe that the current definition of CP is not adequate to provide extended support for robustness. A Category Partition specification is mainly

based on the functionality of the system under test where limited number of Error choices can be introduced for out of bound testing. In order to provide extended support for robustness, the Category Partition technique can be extended by introducing specifications (categories and choices) based on robustness testing of the system similar to the current definition where specifications are based on functionality. The extent of robustness testing of the system will undoubtedly depend of the criticality of the system and the expertise of the tester.

- We used three combinational testing tools for the generation of test frames. We conclude that CASA, which is based on a metaheuristic technique, generates the same or a smaller number of test frames than ACTS and PICT, which are based on a greedy algorithm. Additionally, the time taken by CASA to generate test frames is greater than the time taken by ACTS or PICT, which is not surprising, though execution times are not detrimental to the use of the tools. This was evident during the generation of test frames for the industrial case study.
- While comparing the quality of the test suites when all kinds of constraints are satisfied, to the test suites when only Single and/or Error annotations are used, we conclude that the results do not have major differences. The mutation score, code coverage, test suite cost and number of effective test cases in both cases differ by only up to ±13% approximately.

To conclude, briefly, for software with or without good error handling code, specifying all kinds of constraints, i.e., Single, Error (for error handling code) and Constraints among choices, certainly results in better quality (better fault detection, code coverage and reduced test suite cost) than without using these constraints at all. However,

if a test engineer has limited resources and the functionality of the system allows her to annotate Single and Error choices, meticulously, she is likely to obtain an equally effective test suite with not more than approximately 13% reduction in quality.

Chapter 8 —An Extension of Category Partition Testing for Highly Constrained Systems

Generating test frames with the Each Choice and Pairwise criteria is akin to 1-way and 2-way combinatorial testing [11]. Constraints ensure that only valid combinations of inputs are given to the system under test. A lot of literature is available on constraint handling in combinatorial testing [15, 20, 41, 53, 54], as a result, a number of tools and algorithms are available for supporting constraints during test frame generation [16]. Grindal et. al. [55] also proposed some constraint handling mechanism used with the t-way coverage criterion but there is no mention of constraint handling for the Base Choice criterion.

Grindal [13] mentioned that Base Choice has been found to detect different types of faults as compared to Pairwise. This is essentially because of domain knowledge that leads to the identification of base choices. Grindal also mentioned that Base Choice performed better when there were limited number of candidates for a base choice in a category. Although Base Choice looks promising, there is a lack of literature on how to handle complex constraints, Ammann and Offutt being the exception.

Ammann and Offutt [1, 6] proposed a method for handling infeasible combinations during the generation of test frames using Base Choice. If an infeasible combination is found in a base choice test frame then they suggest to change a conflicting choice with some other choice of the category so that the test frame becomes a feasible combination. They illustrate their idea on a simple example containing two conflicting choices. We find their procedure is not precise enough and there is no guarantee it would work on

more complex constraints (we will actually show using examples that this statement is true). This motivated us to improve the definition of the Base Choice criterion, which resulted in two different flavours of the criterion.

To summarize, there is no published work, to the best of our knowledge, that can help someone deal with constraints on choices in a systematic way when producing a Base Choice adequate test suite in the context of Category Partition.

Contrary to the Each Choice and Pair-Wise criteria, which, similarly to many other selection criteria (e.g., structural ones), have a descriptive definition, the Base Choice criterion has a prescriptive definition [1]. Indeed, the former tell us which test objectives to achieve without telling us how to do that whereas the latter tells us how to proceed to obtain a set of adequate test frames. Specifically [1], given the specification of a base choice for each category, the Base Choice criterion proceeds as follows. A first test frame, which we refer to as a base test frame, is obtained by selecting base choices for all the categories. Given base choices indicate the most likely situations (for categories) that would be exercised during a typical execution of the system under test, this first test case is somewhat a quintessential test case. This base test frame is therefore assumed to be feasible, i.e., it is possible to find input values to concretize it. The subsequent test frames are obtained by changing one base choice to a non-base choice per test frame at a time until all the non-base choices are used at least once in test frames. The changes from a base choice to a non-base choice is always made in the base test frame ensuring that the new test frame is as close as possible to the base choices. We note that, the intent is therefore to ensure that the Base Choice criterion subsumes the Each Choice criterion, i.e., any Base Choice adequate test suite is also Each Choice adequate. Those additional

test frames are therefore slight changes from the initial one. Further, the original definition of the criterion allows slight deviations to the requirement that only one base choice be changed into a non-base choice for that category when creating test frames from the base test frame: if two choices in a test frame are constrained, subsequent test frames can be obtained by moving away from the base choices for more than one category in a test frame, while accommodating constraints. We note that when there are constraints on choices, the Base Choice criterion does not necessarily ensure that each choice is exercised in the set of test frames which is somewhat counter-intuitive [11].

This Chapter presents two criteria, inspired by Base Choice, to solve these issues. Given the prescriptive nature of the Base Choice criterion definition, it is not surprising that our two extensions of the criterion are also prescriptive. Before implementing the criteria the constraints on each choice are converted to CNF, i.e., conjunction of disjunctions as discussed in section 4.3.5. For example, the constraint [!(p22&&p51) && !(p21&&p52)] on a specific choice will be changed to [(!p22 || !p51) && (!p21 || !p52)] which is a logically equivalent conjunction of disjunctions.

We next discuss these two new criteria (sections 8.1 and 8.2) and then illustrate them on an example (section 8.3). We conclude this Chapter in section 8.4.

8.1 Constrained Base Choice (CBC)

Similarly to the Base Choice (BC) criterion, the Constrained Base Choice (CBC) criterion starts with the creation of the first test frame that combines all the base choices (step 1 in Figure 8-1): the base test frame. It is worth noting that BC, and therefore CBC thereby assume that such an initial test frame is feasible, i.e., one can find test inputs that satisfy that test case specification. Although nothing can be found about this assumption

Step 1	Make the first test frame with all the base choices ensuring the constraints are satisfied.
Step 2	To obtain the next test frame change one (base) choice of a category to a non-base choice while keeping all the other (base) choices unchanged.
Step 3	Check the feasibility of the newly created test frame, i.e., the combination of newly selected non-base choice with the other base choices and their associated constraints.
Step 4	If the test frame is feasible, move to step 6.
Step 5	If this test frame is infeasible, i.e., the constraints of its choices conflict with each other, minimally change the other base choices to accommodate the new non-base choice by repeatedly applying the following steps until there is no conflicting choices:
Step 5.i	If a constraint leading to infeasibility contains an **OR (disjunct)** between choices, satisfy any one clause in the disjunction by changing one base choice in the clause into a non-base choice that resolves the conflict. Select that clause that results in minimum changes.
Step 5.ii	If a constraint leading to infeasibility contains **an AND (conjunct)** between choices, satisfy all the clauses involved in the conjunction by changing as few base choices as possible into non-base choices such that the conflict disappears.
Step 5.iii	If a constraint leading to infeasibility contains **conjunctions of disjunctions (CNF)** between choices, proceed as in step 5.i for clauses involved in disjunction and step 5.ii for clauses involved in conjunction.
Step 6	Follow steps 2-5 repeatedly until all the non-base choices have been involved at least once in the set of test frames.
Step 7	This procedure may create duplicate test frames, which needs to be removed from the set of test frames.

Figure 8-1: Constrained Base Choice (CBC) adequate test suite construction procedure

in literature, we believe it is a reasonable one since base choices are set to represent the value of a category that would most likely occur when the program executes. This is an assumption that we have confirmed with academic examples we have found in the literature as well as in case studies provided by our industry partner.

The procedure to create a CBC adequate test suite is iterative (steps 2 to 6 in Figure 8-1) and, if no conflicting choice combination is encountered (this is checked at step 3), then the result is identical to the original BC criterion. In case conflicting choices in a test frame are detected (step 5), then such conflicts are solved in an iterative manner: steps 5.i, 5.ii and 5.iii. Since the newly created test frame (step 2), with conflicting choices as identified in step 5, has been created by changing one base choice into a non-base choice from the feasible base test frame, unfeasibility is necessarily due to the constraint on this non-base choice or a constraint on a base choice. Regardless, such a constraint, which leads to unfeasibility, involves that non-base choice as well as at least one base choice; and there might be more than one such constraint, in which case more than one change may be required, thus the possible repeated modifications on step 5 (i.e., its sub-steps).

Satisfying CBC proceeds differently on step 5 depending on whether this constraint is a disjunction (step 5.i), a conjunction (step 5.ii) or a conjunction of disjunctions (step 5.iii) of choices. Since the new test frame contains a non-base choice (step 2), since the base test frame contains all the base choices, since we want CBC to subsume BC, we need to involve more non-base choices. Sub-steps of step 5 therefore change at least one (additional) base choice into a non-base choice of the same category. When there is a disjunction, we only need to make a change in a choice involved in one clause whereas when there is a conjunction we need to change a choice involved in each clause. If the constraint involves a conjunction of disjunctions (CNF) we have to ensure that any one of the choices involved in the disjunction is changed whereas all the choices involved in the conjunction are changed. Regardless, the procedure is to make wise decisions so that as few changes as possible are performed to make the test frame feasible; this typically

requires looking ahead to evaluate the impact of changes (base choices into non-base choices) to adjust decisions and ensuring that there are a minimum deviations from the base choices.

Because the procedure looks at one test frame at a time, trying to solve conflicting choice constraints, and those constraints do not change (they come from the CP specification), changes applied to make different test frames feasible may involve changes to the same choices, which will result in duplicate test frames. Duplicate test frames are removed in step 7. Note that removing duplicates can also happen during the procedure: a new test frame is added to the list of already created test frames unless it is a duplicate of a test frame already in the list of created ones.

This procedure is iterative and the question of its termination arises. In case the loop on steps 2-7 does not terminate, this means that there exist a choice in the CP specification that cannot be combined in any way in a feasible test frame with other choices. This denotes a problem (a fault) in the CP specification. For example, consider the base test frame 14, 23, 34, 41, 103, 152 (Table 8-IV). The first change from ch_152 to ch_151 will need to satisfy constraint (p31 ||p32 ||p33) on ch_151 (Table 8-I). Assume that ch_31, ch_32 and ch_33 further have a constraint (!ch_151). This would be an issue with the CP specification and the test engineer would need to reconsider the constraints. Such situations can easily be detected with the help of SAT solvers. Literature [126] shows that SAT solvers like zChaff [127] can be used to check the feasibility and such solvers work on the CNF format, similar to the format that our algorithm supports.

By construction, CBC subsumes BC and subsumes Each Choice.

8.2 Extended Constrained Base Choice (ECBC)

The procedure to obtain an Extended Constrained Base Choice (ECBC) adequate test suite begins with the construction of a first test frame consisting of all the base choices, similarly to BC and CBC: step 1 in Figure 8-2. One important difference with CBC, right from the construction of the first test frame but also in further steps (see below) is to recognize that in the event a constraint on a choice is a disjunction, several combinations of choices are worth considering, each one satisfying one clause of the disjunction. In such an event, CBC considers only one combination, i.e., one clause, whereas ECBC will consider them all.

Base choices may have constraints that are disjunctions. These disjunctions and corresponding choice combinations are not necessarily exercised in test frames produced by other steps of the ECBC procedure. To solve this issue, the ECBC procedure differs from CBC in step 2, where we select a base choice and consider its constraint. If there is no constraint or the constraint is a conjunction, then the procedure continues with step 3. The beginning is then identical to CBC. If the constraint is a disjunction, we change the base test frame by changing one or more other base choice than the one having a disjunctive constraint into non-base choices such that each disjunction is exercised once.

Similarly to BC and CBC, each base choice in the base test frame will be eventually considered thanks to the iteration over steps 2-6 (see step 7).

Thereafter, we proceed similarly to CBC, by changing the selected base choice (from step 2) to a non-base choice (step 3), checking the feasibility of the resulting test frame (step 4) and in case of an infeasible combination, further iterating over step 6 and its sub steps to produce a feasible test frame. Since ECBC is an extension to CBC, we resolve

constraints similarly to how we did in CBC but use all the possibilities offered by disjunctions rather than looking for a minimal change from the base choices as in CBC. Further, we also assume that the constraints are feasible and a SAT solver can ensure that, similar to an assumption we made in case of CBC.

We have various sub steps in step 6 in order to systematically cover all the possibilities in the disjunction. There can be situations where the constraints leading to infeasibility have choices tied together with the OR disjunct and those choices belong to the same category (step 6.i(a)), i.e., we keep on generating test frames by satisfying all clauses (choices) but one at a time and generating multiple test frames. This is easily achievable because a test frame can only have one choice per category. However, if the constraint has choices from different categories grouped together with a disjunction then we follow step 6.i(b). In this case, we begin by satisfying one clause in the disjunction, at a time (which is doable because of disjunction), and generate test cases by combining the newly changed clause with the other unchanged clauses (belonging to other categories) in the disjunction, until all the clauses in the disjunction are exhausted. This will result in multiple test frames. In case the constraint has a conjunction (step 6.ii) we will proceed similarly to CBC, changing all the clauses and satisfying the conjunction. We also consider the scenario where the constraint is complex and is a conjunction of disjunctions (step 6.iii) (recall that we convert the constraints to CNF format). When the choices involved in a disjunction belong to the same category, we proceed as in step 6.i(a) and simultaneously make combinations with the conjunct clauses, after changing the conjunctive clauses (step 6.ii). A test frame is generated after each change. Similarly, if the choices in a constraint belong to different categories, we proceed as in step 6.i(b) and

make their combinations with the conjunctive clauses after each change, thereby resulting in multiple test frames.

While considering each possibility offered by the disjunctions, we are also resolving the constraints introduced due to the new non-base choices or constraints they have with already existing base choices. In this process, we will move farther away from BC as compared to CBC and at the same time adding more test frames.

The process iterates over steps 2 to 6 (step 7) until all the non-base choices are involved in test frames at least once, followed by a duplicate removal step (step 8).

Similarly to CBC, the ECBC procedure terminates unless there is an issue in the CP specification. ECBC differs from CBC in how it handles disjunctions in constraints. By construction, ECBC subsumes CBC.

We perform the complexity (Big-O) analysis of the ECBC analysis for the worst-case scenario in Appendix E. We discuss the time complexity in the presence of both conjunctions and disjunctions in Appendix E.

Step 1	Make the first test frame with all the base choices ensuring the constraints are satisfied.
Step 2	Select a base choice, which is a candidate for change to a non-base choice, in the base test frame produced in step 1. If the choice does not have a disjunctive constraint, then move to step 3. If the choice has a disjunctive constraint and the disjunction is between the choices of the same category then move to step 6.i(a). If the disjunction is between the choices of the different categories move to step 6.i(b). Satisfying each clause using these steps will result in a feasibility check of that clause and its associated constraints with all the other choices of the test frame. This would generate multiple test frames, one new test frame per change in the disjunction.
Step 3	To obtain the next test frame, change the base choice selected in step 2 to a non-base choice while keeping all the other (base) choices

	unchanged.
Step 4	Check the feasibility of the newly created test frame, i.e., the combination of the newly selected non-base choice with the other base choices, and their associated constraints.
Step 5	If the test frame is feasible, move to step 7.
Step 6	If the test frame is infeasible, i.e., the constraints of its choices conflict with each other, change the other base choices to accommodate the new non-base choice by repeatedly applying the following steps until there is no conflicting choices. Satisfying each clause using these steps will result in a feasibility check of that clause (choice combination) and its associated constraints with all the other (base) choices of the test frame. This can result in changing one or more base choice into a non-base choice in an attempt to satisfy that clause and obtain a feasible test frame.
Step 6.i(a)	If a constraint leading to infeasibility contains an **OR (disjunct)** between choices belonging to the **same category**, consider all the possible ways to make this constraint true by satisfying all the clauses in the disjunction, one at a time, and obtaining multiple test frames.
Step 6.i(b)	If a constraint leading to infeasibility contains an **OR (disjunct)** between choices belonging to **different categories**, consider all the possible ways to make this constraint true by satisfying all the clauses in the disjunction, one at a time, and combining the newly changed choice with the existing unchanged choices of the other categories in that constraint until all the choices in the constraint have been changed thereby resulting in multiple test frames.
Step 6.ii	If a constraint leading to infeasibility contains an **AND (conjunct)** between choices of same or different categories, satisfy all the clauses involved in the conjunction by changing associated base choices to a non-base choice, if required to satisfy the constraint; otherwise change a minimum number of choices.
Step 6.iii	If a constraint leading to infeasibility contains **conjunctions of disjunctions (CNF)** proceed as in step 6.i(a)if choices in disjunction belonging to the **same category** and step 6.i(b) if choices in the disjunction belong to different categories. Follow step 6.ii for conjunctive clauses and concurrently make their combinations with the newly changed disjunctive clauses, one at a time, whilst satisfying the

```
                constraints, which will result in multiple test
                frames.
Step 7      Follow steps 2-6 repeatedly until all the non-base
              choices have been involved at least once in the
              set of test frames.
Step 8      This procedure may create duplicate test frames,
              which need to be removed from the set of test
              frames.
```

Figure 8-2 Extended Constrained Base Choice (ECBC) adequate test suite construction procedure

8.3 Illustrative Examples

The newly proposed selection criteria are illustrated using two examples. Example 1 is a subset CP specification obtained from industrial case study CS_L2_A and example 2 is a subset of the NextDate case study. The test frames in tables are color coded for better clarity. The yellow background color depicts the categories whose base choice is flipped to a non-base choice in order to obtain new test frames. The test frames in pink background are the feasible test frames obtained using the base choice criterion. These test frames have choices that do not require repair using the CBC and ECBC criteria because a change from a base choice to a non-base choice, in order to obtain the next test frame, does not introduce any conflict. The choices in the blue background are changes from a base choice resulting from the constraints in the base choice or in the non-base choice. Choices in green background depict the choices, and hence the test frames, which are introduced due to disjunctions in constraints; this is specific to ECBC.

8.3.1 Example 1

The Example 1 CP specification has five parameters, six categories, 23 choices, nine of which have a constraint (Table 8-I). Parameter 1 has two categories (Cat 1 and Cat2), which have four and three choices, respectively. The base choices are ch_14 (for Cat 1), ch_23 (Cat 2), ch_34, ch_41, ch_103, ch_152: they are highlighted in red in the table.

Table 8-I is an example CP specification to illustrate the BC, CBC and ECBC selection criteria in the presence of complex constraints.

8.3.1.1 Test frames using the Base Choice criterion

Table 8-II shows the test frames one obtains when using the Base Choice criterion. The initial implementation of BC (assuming that there are no constraints among choices) will result in 18 test frames. This number is based on the total number of choices per category and can be obtained from the formula mentioned by Grindal et. al. [7], i.e., the number of test cases in a BC adequate test suite is equal to 1+ $\sum_{i=1}^{N}(v_i - 1)$, where N is the number of categories and v_i is the number of choices in category i.

Table 8-I: CP specification for example 1

	Parameters	Categories	Choices	Properties	Constraints
1	P1	Cat 1	ch_12	p12	
			ch_13	p13	
			ch_14	p14	
			ch_15	p15	
		Cat 2	ch_21	p21	if (p12 || P13) && p41 && (p37 ||p38 ||p39)
			ch_22	p22	if (p12 ||p13||p14) &&(p42) &&(p37 ||p38 ||p39)
			ch_23	p23	If (p15 || p14) &&(p15 || !p42)
2	P2	Cat 3	ch_31	p31	
			ch_32	p32	
			ch_33	p33	
			ch_34	p34	
			ch_36	p36	
			ch_37	p37	
			ch_38	p38	
			ch_39	p39	
3	P3	Cat 4	ch_41	p41	
			ch_42	p42	
4	P7	Cat 10	ch_101	p101	if (p31)
			ch_102	p102	if (p32|| p33)
			ch_103	p103	if (p32 ||p34||p36||p37||p39)
			ch_104	p104	if (p32 ||p38)
5	P11	Cat 15	ch_151	p151	If (p31||p32||p33)
			ch_152	p152	if (p34||p36||p37||p38||p39)

Table 8-II Test frames using Base choice criterion containing feasible frames

TF	Cat1	Cat2	Cat3	Cat4	Cat10	Cat15	Frame Type
1	14	23	34	41	103	152	Feasible
2	14	23	34	41	103	151	Infeasible
3	14	23	34	41	101	152	Infeasible
4	14	23	34	41	102	152	Infeasible
5	14	23	34	41	104	152	Infeasible
6	14	23	34	42	103	152	Infeasible
7	14	23	31	41	103	152	Infeasible
8	14	23	32	41	103	152	Infeasible
9	14	23	33	41	103	152	Infeasible
10	14	23	36	41	103	152	Feasible
11	14	23	37	41	103	152	Feasible
12	14	23	38	41	103	152	Infeasible
13	14	23	39	41	103	152	Feasible
14	14	21	34	41	103	152	Infeasible
15	14	22	34	41	103	152	Infeasible
16	12	23	34	41	103	152	Infeasible
17	13	23	34	41	103	152	Infeasible
18	15	23	34	41	103	152	Feasible

The first test frame in Table 8-II consists of all the base choices. The subsequent test frames are obtained by changing one base choice at a time to a non-base choice, as shown by TF2, TF3 etc. The choices shown in yellow background color are changes from base choice to a non-base choice thereby resulting in additional test frames. Assuming that we start changing a base choice to a non-base choice starting from category 15, this technique will add one test frame for category 15 (because category 15 has only one non-base choice), three test frames for category 10 (because it has three non-base choices), one test frame for category 4, seven test frames for category 3, two test frames for category 2 and three test frames for category 1, resulting in a total of 18 test frames.

Applying the constraints shown in Table 8-I, only five out of the 18 test frames are feasible. Thirteen test fames are infeasible because the change from a base choice to a non-base choice results in a conflict with the existing base choices: constraints on choices involved in the test frames are not satisfiable together, i.e., it is not possible to find test

inputs for such test frames. For example, consider changing base choice ch_152 of Cat 15 to non-base choice ch_151. This results in an infeasible test frame because the constraint on choice ch_151 allows a combination of this choice with ch_31, ch_32 or ch_33 of Cat 3. These choices are all non-base choices and therefore different from already existing choice ch_34, which is already in the test frame (it is the base choice of category 3).

In such a case, following Ammann and Offutt solution [6], to handle infeasible test frames produced in the BC procedure, we can change base choice ch_34 into either non-base choice ch_31, ch_32 or ch_33 to make this change a feasible change and hence a feasible test frame. Ammann and Offutt, however, do not provide guidance as to whether one should choose ch_31, ch_32 or ch_33. In other words, their description lacks guidance on how to select a choice in case of multiple options. Furthermore, obtaining a feasible test frame after such a change is not a guarantee as other constraints may still be unsatisfied. For instance, changing ch_34 into ch_31 will require that we additionally change base choice ch_103 to non-base choice ch_101 to satisfy other constraints, whereas changing ch_34 into ch_32 will result in no further change (i.e., all constraints are satisfied). Ammann and Offutt do not mention the need to look ahead at possible ripple effects on feasibility of base-choice to non-base-choice changes. As a consequence, in our highly constrained situations, we could not decide how to proceed with the implementation of the Base Choice criterion in the presence of complex constraints by following Ammann and Offutt solution. Therefore in the given circumstances, we are relying on one possible implementation of BC [11] where the infeasible test frames, i.e., the frames that do not satisfy the constraints mentioned in Table 8-I, are removed. There are five feasible test frames that are combined using the

Base Choice criterion suggested by Amman and Offutt (Table 8-II) and some choices (e.g., ch_151, ch_101, ch_102, ch_104, ch_42 etc.) corresponding to infeasible test frames are not exercised. All the infeasible test frames in Table 8-II will be repaired in section 8.3.1.2 and 8.3.1.3 using the CBC and ECBC criteria.

8.3.1.2 Test frames using the Constrained Base Choice (CBC) Criterion

Table 8-III shows the test frames obtained using the CBC selection criterion. The first step when creating a CBC adequate test suite (Figure 8-1) is to create a base test frame, i.e., a test frame with only base choices. This is test frame (TF) 1 in Table 8-III. Remember that since the selection of base choices is based on the tester's expertise, we assume that this first test frame is feasible. On the example CP specification, we need to iterate over steps 2-6 (Figure 8-1) 17 times, as discussed below.

In the first iteration, starting from the last category (because it has an interesting constraint, useful to illustrate CBC), we change ch_152 with the only non-base choice of the category, namely ch_151 (step 2 of Figure 8-1). We obtain the combination of

Table 8-III: Application of CBC on Example 1

TF	Cat1	Cat2	Cat3	Cat4	Cat10	Cat15
1	14	23	34	41	103	152
2	14	23	32	41	103	151
3	14	23	31	41	101	151
4	14	23	32	41	102	151
5	14	23	38	41	104	152
6	15	23	34	42	103	152
7	14	23	33	41	102	151
8	14	23	36	41	103	152
9	14	23	37	41	103	152
10	14	23	39	41	103	152
11	14	22	37	42	103	152
12	12	21	37	41	103	152
13	15	23	34	41	103	152
14	13	21	37	41	103	152

choices: ch_14, ch_23, ch_34, ch_41, ch_103, ch_151. Checking feasibility (step 3), we notice the constraint on ch_151 indicates it can only co-exist with ch_31 OR ch_32 OR ch_33 (all choices of category 3); unfortunately, the newly created combination involves ch_34 (also from category 3), which is a base choice but makes this combination infeasible. We need to run step 5, in fact step 5.i because the constraint on ch_151 is a disjunction. The base choice for category 3 is ch_34, which needs to be changed into either one of ch_31, ch_32 or ch_33 to make the choice combination feasible. The selection of the non-base choice should be such that it should lead to minimal changes to the test frame. Let us investigate the selection of any of these three choices one after the other.

Suppose we replace ch_34 with ch_31: the test frame becomes ch_14, ch_23, ch_31, ch_41, ch_103, ch_151. The next step is to check the feasibility of this test frame. We observe that this test frame is infeasible because of the constraint on ch_103, which cannot be combined with ch_31 (only with ch_32, ch_34 …). Selecting ch_31 requires changing another base choice, i.e., ch_103, into a non-base choice, specifically ch_101, thereby resulting in two base choice to non-base choice changes. Test frame ch_14, ch_23, ch_31, ch_41, ch_101, ch_151 is feasible though.

If instead we select ch_32, the test frame becomes ch_14, ch_23, ch_32, ch_41, ch_103, ch_151. We observe that the test frame is feasible. Only one base to non-base choice change is required to obtain this feasible test frame.

Last, if we select ch_33, the test frame becomes ch_14, ch_23, ch_33, ch_41, ch_103, ch_151. Choices ch_33 and ch_103 cannot be combined, making this test frame

infeasible; we need to change ch_103 into ch_102. Similarly to the first alternative, we obtain a feasible test frame with two changes.

Considering the three alternatives, selecting ch_32 is the change selected in step 5.i because it results in a minimal number of changes to base choices. This iteration leads to test frame 2 (Table 8-III).

In a second iteration of steps 2-6, we change (step 2) the base choice of category 10 (ch_103) into a non-base choice, such as ch_101. We obtain test frame ch_14, ch_23, ch_34, ch_41, ch_101, ch_152. Since ch_101 can only co-exist with ch_31 (constraint on ch_101), the test frame is infeasible (step 3), and we execute step 5.i (only one clause in the constraint). We replace ch_34 with ch_31. After making the suggested change to ch_31 we check the feasibility of the test frame. We observe that ch_31 cannot be combined with ch_152 and therefore the infeasibility still persists: ch_14, ch_23, ch_31, ch_41, ch_101, ch_152 is infeasible. We change ch_152 into ch_151. We again check the feasibility of the test frame. We have obtained a feasible test frame: the third test frame in Table 8-III.

Similarly, iterations 3 and 4 of steps 2-6, which change ch_103 into ch_102 and ch_104, respectively, will lead to feasible test frames 4 and 5 (Table 8-III).

In iteration 5, base choice ch_41 of category 4 is changed to ch_42 (the only non-base choice). After making this change we check the feasibility of the test frame (step 3). We observe that the test frame ch_14, ch_23, ch_34, ch_42, ch_103, ch_152 is infeasible as ch_42 is constrained with ch_23 (step 4). We execute step 5.iii because the constraint involves a conjunction of disjunctions. For both the clauses in the conjunction (p15 || p14) && (p15 || !p42) we follow step 5.ii and for individual clauses in the disjunction we

follow step 5.i. In order to satisfy the second clause using step 5.i we satisfy p15 as we cannot change p42 because that is the current non-base choice we are dealing with. Because of that, the only possible change for the first clause is p15 so we change ch_14 into ch_15 to obtain a feasible test frame. This change results in satisfying both the clauses and we obtain feasible test frame 6 (Table 8-III).

For iterations 6 to 12, we change the choices of category 3 from base choice ch_34 to non-base choices ch_31, ch_32…ch_39. Changing the base choices to ch_31, ch_32 and ch_38 results in duplicate frames similar to test frames 3, 2 and 5 respectively. Test frames 7 to 10 are obtained using the same procedure mentioned above.

In iteration 13, we change the base choice ch_23 to non-base choice ch_22 (step2). The feasibility check (step 3) discovers that ch_22 is constrained with (p12 ||p13 ||p14) && (p42) && (p37 ||p38 ||p39). Therefore, the test frame ch_14, ch_22, ch_34, ch_41, ch_103, ch_152 is infeasible. Since the constraint is a conjunction of disjunctions (step 5.iii) we execute step 5.i to satisfy either one of the disjunctive clause and step 5.ii to satisfy all the conjunctive clauses. The first clause, involving choices of category 1 does not require a change from a base choice to a non-base choice as one element of the disjunction ch_14 is a base choice (any one element should satisfy). For the second clause, we change the base choice ch_41 to a non-base choice ch_42 (step 5.ii). We further check the feasibility of ch_42 with the constraints of all the other choices. It is feasible, therefore, ch_42 is selected. In the third clause, none of the choices (ch_37, ch_38, ch_39) is a base choice, therefore ch_34 should be changed to any one of them to get a feasible test frame (step 5.i).

Suppose we select ch_37 to satisfy this part of the constraint. Checking the feasibility of ch_37 with all the other constraints, we discover that the test frame is feasible. Selecting choice ch_38 would result in a change of ch_103 into ch_104, and selecting ch_39 would result in no other change. We can select either ch_37 or ch_39: we select ch_37 and we obtain test frame 11 (Table 8-III).

Similarly, from iterations 14 we can obtain test frame 12. Iteration 15, 16 and 17 will result in test frames 13 and 14 respectively (Table 8-III). One of the test frame, obtained by changing ch_14 to ch_12 is removed, since it is a duplicate of test frame 12.

The total number of frames obtained using CBC for this example is 14.

8.3.1.3 Test frames using the Extended Constrained Base Choice (ECBC) criterion

This section illustrates ECBC: Figure 8-2. As mentioned earlier ECBC is an extension to CBC, begins similarly but takes better care of choice constraints that are disjunctions. We discuss only those steps that are specific to ECBC. The rest of the steps proceed similarly to CBC. The resulting test frames are in Table 8-IV.

The first test frame is the base test frame, which is a combination of all the base choices (step 1). In step 2, we select a base choice which is a candidate to be changed into a non-base choice. Assume we select ch_152. Before we actually change it to a non-base choice, we consider its constraint and observe (Table 8-I) it is a disjunction between the choices of the same category: p34 ||p36 ||p37 ||p38 ||p39. Step 2 is to involve each of the choices in each clause of the disjunction into a new test frame. Since these are all choices of category 3 (Table 8-I) we follow step 6.i(a). This category has base choice ch_34, each new test frame replaces ch_34 with each of the other four non-base choices while at the

same time checking the feasibility of the change with other choices. For instance, while changing ch_34 to ch_38 we observe that ch_38 is constrained with base choice ch_103 and is only possible with ch_104. Therefore, we make this change and further check the feasibility of ch_104 with other base choices. This results in test frames 2, 3, 4 and 5, which is the change from base choice ch_34 of category 3 to ch_36, ch_37, ch_38 and ch_39, respectively (Table 8-IV).

Next, step 3 changes the base choice selected in step 2 (ch_152) to a non-base choice: it can only be ch_151 (Table 8-I). The test frame is therefore ch_14, ch_23, ch_34, ch_41, ch_103, ch_151. We check the feasibility of the test frame (step 4), which

Table 8-IV: Application of ECBC on Example 1

TF	Cat1	Cat2	Cat3	Cat4	Cat10	Cat15
1	14	23	34	41	103	152
2	14	23	36	41	103	152
3	14	23	37	41	103	152
4	14	23	38	41	104	152
5	14	23	39	41	103	152
6	14	23	31	41	101	151
7	14	23	32	41	103	151
8	14	23	33	41	102	151
9	14	23	32	41	102	151
10	14	23	32	41	104	151
11	15	23	34	42	103	152
12	15	23	34	41	103	152
13	14	22	37	42	103	152
14	14	22	38	42	104	152
15	14	22	39	42	103	152
16	12	22	37	42	103	152
17	12	22	38	42	104	152
18	12	22	39	42	103	152
19	13	22	37	42	103	152
20	13	22	38	42	104	152
21	13	22	39	42	103	152
22	12	21	37	41	103	152
23	12	21	38	41	104	152
24	12	21	39	41	103	152
25	13	21	37	41	103	152
26	13	21	38	41	104	152
27	13	21	39	41	103	152

means we consider the constraint associated to ch_151: ch_31 ||ch_32 ||ch_33. Since the test frame contains ch_34 and ch_151 cannot be combined with it, the test frame is infeasible. Since the constraint is a disjunction with the choices of the same category, we run step 6.i(a). Unlike CBC where we select one disjunctive clause, here we make multiple test frames each one involving one of the clauses in the disjunction. We look at each clause of the disjunctive constraint to produce several test frames.

We proceed by selecting ch_31 and the test frame becomes: ch_14, ch_23, ch_31, ch_41, ch_103, ch_151. We check its feasibility and observe the constraint on ch_103 specifies it cannot be combined with ch_31: the choice combination is infeasible. Therefore, in order to accommodate this constraint, we further change ch_103 to ch_101 and obtain the test frame: ch_14, ch_23, ch_31, ch_41, ch_101, ch_151. Checking the feasibility of this frame we observe that it is feasible and we obtain test frame 6.

We now select the second disjunctive clause: ch_32. We get the test frame ch_14, ch_23, ch_32, ch_41, ch_103, ch_151. This test frame is feasible; this is test frame 7.

On selecting the third disjunctive clause, ch_33, the test frame becomes ch_14, ch_23, ch_33, ch_41, ch_103, ch_151. On checking the feasibility of this test frame we observe that the constraint on ch_103 does not accept ch_33. Ch_33 can only exist with choice ch_102, which is not a base choice. Therefore, ch_103 is changed to ch_102 and we obtain the test frame ch_14, ch_23, ch_33, ch_41, ch_102, ch_151. We check the feasibility of this test frame and observe no conflict. Therefore, we obtain test frame 8. This ends the first iteration of steps 2-6.

Similarly, we consider the base choice of category 10 (step 2), i.e., ch_103 and observe its constraint is a disjunction with choices of the same category: (p32 ||p34 ||p36

||p37 ||p39). On creating test frames while satisfying the disjunction, we observe that these test frames are already covered in the previous iteration and therefore we do not add them. (Note here that we illustrate the removal of duplicates during the iteration rather than at the very end, in step 8.) We then (step 3) change the base choice ch_103 to non-base choice ch_101 and observe that the resulting test frame it again a duplicate, which is therefore not added. Ch_103 is next changed to ch_102. Having two clauses in the disjunctive constraint for ch_102 (step 6.i (a)) we change the base choice ch_34 to ch_32 and in order to satisfy the otherwise infeasible choice combination, we change ch_152 to ch_151. We obtain test frame 9: ch_14, ch_23, ch_32, ch_41, ch_102, ch_151. The test frame created by changing ch_34 to ch_33 is identical to test frame 8.

Assume a change from base choice ch_23 to non-base choice ch_22 and ch_21 to form the next set of test frames. Before making a change from a base choice to a non-base choice we begin by step 2 and check the constraints on ch_23. The choice has constraints which is a conjunction of disjunctions and has choices from the same as well as different categories (step 6.iii), i.e., (p15 || p14) && (p15 || !p42) . We follow step 6.i (a) for the first clause in the conjunction (p15 || p14) and satisfy both clauses in the disjunction, one by one, and form a test frame. Considering ch_14, we observe that it is already a base choice and is satisfied (TF1). We change ch_14 to ch_15 and form a test frame; we get ch_15, ch_23, ch_34, ch_41, ch_103, ch_152 (TF 12). By making this change we observe that we are also satisfying the second clause in the conjunction (p15||!p42). We now have to consider all possible combinations in the second clause as per step 6.i (b) and satisfy the disjuncts, one at a time. Currently both the clauses are satisfied. TF1 satisfies the second clause and not the first clause and as the third

combination, we satisfy the first clause and not the second clause and make a combination of p15 with p42. Therefore, we obtain a test frame ch_15, ch_23, ch_34, ch_42, ch_103, ch_152, which is test frame 11.

Changing base choice ch_23 to ch_22 we observe that ch_23 is constrained with a conjunction of disjunctions (step 6.iii); (p12 ||p13 ||p14) &&(p42) &&(p37 ||p38 ||p39). We will first begin by satisfying the choice that is alone in the term of the CNF, which is p42, and change base choice ch_41 to ch_42. Checking the feasibility of ch_42 with all other base choices we observe that it is feasible. Now for the choices involved in the disjunction, we follow step 6.i(a). We will consider each choice in the disjunction and make a test frame while not changing every other choice. Consider the first choice in the disjunction, i.e., ch_12, therefore making a combination ch_12, ch_22, ch_37 (satisfying the third conjunctive clause), ch_42, ch_103, ch_152. We check the feasibility of this test frame and it is feasible (test frame 16). Now we change ch_12 to the next disjunctive clause ch_13, keep ch_37 un-changed, and check the feasibility, we get feasible test frame 19. We change to ch_14 and get feasible test frame 13. In the next set of test frames we change ch_37 to ch_38 and make combinations again with ch_12, ch_13 and ch_14, we will get test frames 14, 17 and 20 after performing the feasibility check. We do the same for ch_39.

The steps from 2 to 6 are recursively followed to obtain 27 test frames: Table 8-IV. Analyzing the test frames in Table 8-III and Table 8-IV we observe that all the test frames generated for CBC are also part of the ECBC test suite since ECBC differs from CBC in how it handle disjunctions in constraints.

8.3.2 Example 2

We would again illustrate CBC and ECBC using a partial CP specification shown in Table 8-V containing 4 categories and 12 choices; this is a subset of the CP specification of the NextDate case study in appendix B.1. We have ignored the error choices in this partial specification since they are added at the end using the Single Error Choice criterion. The table shows constraint on choice ch4.4, ch4.5 and ch4.6. Constraints specify for instance that if ch4.4 is combined with ch2.2 then ch3.3 should not be present in that test frame and if ch4.4 is combined with ch3.3 then ch2.2 shouldn't be present in that test frame.

Table 8-VI shows the result of test frame generation using the Base Choice criterion (BC) and the two extensions we introduced earlier: CBC and ECBC. The first test frame of the BC set is only made of base choices. The second, third and fourth test frames are copies of the first one except that the second choice is a non-base choice for category number 3. When changing the base choice for category 2 into a non-base choice, keeping

Table 8-V: Partial CP specification of NextDate Case study.

Parameters	Categories	Choices	Constraints
Year	Range	[ch1.2] 1582<=Year<=2100	
	Type	[ch2.1] isLeapYear	
		[ch2.2] isCommonYear	
Month	Length	[ch3.1] 30Days (4, 6, 9, 11)	
		[ch3.2] 31Days(except December) (1, 3, 5, 7, 8, 10)	
		[ch3.3] February (2)	
		[ch3.4] December (12)	
Day	Range	[ch4.1] 1<=Day<28	
		[ch4.4] Day==29	If (!2.2 || !3.3)
		[ch4.5] Day==30	If (!3.3)
		[ch4.6] Day==31	If (!3.3 && !3.1)
		[ch4.7] Day==28	

the other base choices one obtains combination ch2.2, ch3.3, ch.4.4, which is not feasible because of the constraint on choice ch4.4. Similarly, constraints on category 4's choices prevent some combinations. Note that choice ch2.2 does not appear in the set of test frames; BC does not subsume Each Choice. The definition of BC being not precise enough regarding the handling of infeasible test frames because of constraints, the procedure above, whereby an infeasible test frame is simply dropped, is one possibility among others. Ad-hoc procedures to handle such situations can be considered; our contribution is two precise, systematic such procedures.

Table 8-VI: Test frames for Example 2 using BC, CBC and ECBC

BC

1	C 1.2	C 2.1	C 3.3	C 4.4
2	C 1.2	C 2.1	C 3.1	C 4.4
3	C 1.2	C 2.1	C 3.2	C 4.4
4	C 1.2	C 2.1	C 3.4	C 4.4
5	C 1.2	C 2.1	C 3.3	C 4.1
6	C 1.2	C 2.1	C 3.3	C 4.7

CBC

1	C 1.2	C 2.1	C 3.3	C 4.4
2	C 1.2	C 2.2	C 3.1	C 4.4
3	C 1.2	C 2.1	C 3.1	C 4.4
4	C 1.2	C 2.1	C 3.2	C 4.4
5	C 1.2	C 2.1	C 3.4	C 4.4
6	C 1.2	C 2.1	C 3.3	C 4.1
7	C 1.2	C 2.1	C 3.1	C 4.5
8	C 1.2	C 2.1	C 3.2	C 4.6
9	C 1.2	C 2.1	C 3.3	C 4.7

ECBC

1	C 1.2	C 2.1	C 3.3	C 4.4
2	C 1.2	C 2.2	C 3.1	C 4.4
3	C 1.2	C 2.2	C 3.2	C 4.4
4	C 1.2	C 2.2	C 3.4	C 4.4
5	C 1.2	C 2.2	C 3.3	C 4.1
6	C 1.2	C 2.2	C 3.3	C 4.7
7	C 1.2	C 2.1	C 3.1	C 4.4
8	C 1.2	C 2.1	C 3.2	C 4.4
9	C 1.2	C 2.2	C 3.4	C 4.4
10	C 1.2	C 2.1	C 3.3	C 4.1
11	C 1.2	C 2.1	C 3.1	C 4.5
12	C 1.2	C 2.1	C 3.2	C 4.5
13	C 1.2	C 2.1	C 3.4	C 4.5
14	C 1.2	C 2.1	C 3.2	C 4.6
15	C 1.2	C 2.1	C 3.4	C 4.6
16	C 1.2	C 2.1	C 3.3	C 4.7

CBC introduces the combinations of choices that are missing with BC while at the same time satisfying constraints. For instance, in test frame 2, changing base choice ch2.1 to choice ch2.2 is only possible if the constraint on ch4.4 is satisfied, which is achieved by replacing ch3.3 with ch3.1 or ch3.2 or ch3.4 or changing ch4.4 to ch4.1 or ch4.7. The decision is made such that minimum changes are done while satisfying the constraints. So

any one element in the OR (disjunct) constraints is a candidate. We chose ch3.1. CBC has a larger number of test frames than BC, as expected.

ECBC produces as many alternative outcomes as possible when there are disjunctions between choices, as opposed to CBC that selects only one alternative, which results in minimum changes. ECBC proceeds similar to CBC except that we select a base choice that has to be converted to a non-base choice (step 2, Figure 8-2) and check if that choice is involved in any disjunctive clause. We begin by selecting ch2.1 and this choice is not involved in any constraints. We then change ch2.1 to ch2.2 and check the feasibility of the test frame. The test frame ch1.2, ch2.2, ch3.3, ch4.4 is infeasible and the constraint on ch4.4 needs to be satisfied. We can use either ch3.1, ch3.2, or ch3.4 instead of base choice ch3.3 which will give us test frames 2, 3 and 4. Further, we will also use ch4.1 and ch4.7 and do not change ch3.3 and obtain test frames 5 and 6.

Thereafter, we change the base choice ch3.3 to non-base choices ch3.1, ch3.2 and ch3.4 and obtain test frames 7, 8 and 9. Since there are no constraints, these test frames are feasible.

Next, we consider the choices of category 4. Before changing the base choice ch4.4 to non-base choices we first consider the constraints involved with choice 4.4. This choice involves a disjunctive constraint: !ch2.2||!ch3.3. This constraint can be elaborated as C2.1||C3.1||C3.2||C3.4. Since this constraint involves choices belonging to two different categories (step 6.i(b)), we have to satisfy different clauses of the disjunction, one at a time, and form multiple test frames. The Base test frame (TF1) already satisfies C2.1. Therefore, we will now begin by satisfying !C3.3 and not satisfying !C2.2 and we get test frames identical to frames 2, 3 and 4 which will be removed because of

duplication. We then change the base choice ch4.4 to other non-base choices and form test frames 10 to 16. We obtained 16 test frames for ECBC. As expected, ECBC produces more test frames than CBC and subsumes CBC, which subsumes BC.

8.4 Conclusion

The original definition of the Base Choice criterion does not account for complex constraints. We extended this definition in two different ways, referred to as Constrained Base Choice (CBC) and Extended Constrained Base Choice (ECBC), to combine choices strategically similar to the original definition of BC while considering constraints. Briefly, the choices affected by constraints are also changed to their respective non-base choice in order to satisfy the constraints. There can be scenarios where the choices have complex constraints: e.g., ORs among the choices. CBC selects any one candidate among the ORs such that it results in minimum changes to other base choices. ECBC however selects all the OR candidates and forms test frames. By construction, CBC subsumes BC and ECBC subsumes CBC. In the next chapter, we empirically compare the proposed selection criteria with already existing criteria using academic and industrial case studies.

Chapter 9 —Comparison of CBC and ECBC with Existing Selection Criteria

In this chapter, we compare the quality of the test suites obtained using the CBC and ECBC criteria with the test suites obtained using other criteria like Base Choice, Each Choice etc. For the purpose of comparison, we obtain test frames using two covering array generation tools for generating test frames for the Each Choice, Pairwise and 3-way selection criteria. The chosen tools, CASA and ACTS, use different technologies for the generation of test frames; metaheuristic and greedy technique. Both techniques have their pros and cons as we discussed in Chapter 4. We did not want to rely on only one technique for the generation of test frames and their further comparison with CBC and ECBC. Rather we are interested in using both technologies, obtain two different test suites, determine their quality and select the best quality test suite to compare it with the test suite obtained using CBC and ECBC. We in fact went a step further. Realizing that a basic covering array generation tool does not employ single annotation, we used both tools to generate test frames in the presence and in the absence of single annotations. This way we want to compare our proposed criteria with the results akin to a combinatorial testing tool like behaviour (without using single annotations) as well as with the behaviour which is nearer to Category Partition (using single annotations). This will result in four test suites for each selection criterion. We will then compare the best of those four test suites with the results of CBC and ECBC.

In the next section, we begin by formulating research questions (section 9.1) and devise experiments to answer them (section 9.2). We discuss results of individual

experiments in section 9.3 and summarize answers to research questions in section 9.4. We conclude in section 9.5.

9.1 Research Questions (RQs)

To compare the new criteria (i.e., CBC and ECBC) with existing selection criteria (i.e., BC, EC, Pairwise and 3-way), we identify four research questions. The comparison will be done using the experiments proposed in section 9.2.

RQ1: How do CBC and ECBC perform as compared to the Base Choice criterion?

RQ2: How do CBC and ECBC perform as compared to the Each Choice criterion?

RQ3: How do CBC and ECBC perform as compared to the Pairwise criterion?

RQ4: How do CBC and ECBC perform as compared to the 3-way criterion?

RQ1 will be answered by comparing the Base Choice criterion while accounting for constraints using CASA integrated with Melba [11] with CBC and ECBC. RQ2 will be answered by comparing the Each Choice criterion (1-way) with CBC and ECBC. The test frames using the Each Choice criterion will be obtained using CASA and ACTS, both in the presence and in the absence of single annotations, as explained further in section 9.2. Similarly, RQ3 (resp. RQ4) will be answered by comparing a pairwise (resp. 3-way) adequate test suite generated with CASA and ACTS in the presence and in the absence of single annotations, with CBC and ECBC. In order to answer RQ2, RQ3 and RQ4 we are comparing CBC and ECBC with the best result obtained from the four executions of CASA and ACTS in order to make assertive conclusions.

9.2 Design of Experiments

In this section, we discuss the experiments we put together to compare the quality of the test suites obtained using CBC and ECBC with the test suites obtained using existing criteria like Base Choice, Each Choice etc.

For the purpose of comparison, we will obtain test frames using two covering array generation tools CASA [20] and ACTS [19]. Besides using different technologies, i.e., metaheuristic and greedy algorithm, for the generation of test frames, both tools have pros and cons [16]. In our context, experimenting with both tools is interesting because they have different capabilities and characteristics that, we suspect, may influence the test frames they generate. Realizing that a basic covering array generation tool does not employ single annotation, we used both tools to generate test frames in the presence and in the absence of single annotations. In doing so we study two other points of variation in the semi-automated support of CP, which is another contribution of this work: specifically, the tool used to automatically generate test frames, the presence or not of single constraints. In our previous study [128] discussed in Chapter 7, we were able to prove successfully with experimentation that annotation of choices as single results in a better quality test suite. Therefore, in this work we are comparing our criteria with both versions of CP specifications. This will result in four test suites for each selection criterion. This will be followed by selecting the best quality test suite and its comparison with the results of CBC and ECBC.

9.2.1 Experimental setup

We used three academic case studies: NextDate, Triangle and PackHexChar (detailed in section 5.1); and three industry case studies CS_L1_A, CS_L1_C and

CS_L2_A (detailed in section 5.2). We chose these three industrial case studies out of the total 13 because of the complexity of their CP specification. The CP specifications of these case studies have constraints among choices and single annotations. The constraints among the choices would help us see the impact of CBC and ECBC as the results would differ from the generic Base Choice criterion. Further, we can also see the impact of adding and ignoring single annotations for these case studies. We generated two versions of CP specifications for each case study and both versions differ by single annotations, i.e., one version had the required choices annotated as single whereas in the other version the 'could-be' single choices are not annotated as such. Four of the six case studies have two versions of CP specifications. PackHexChar and CS_L2_A do not have single annotations for any choice in the CP specification. Table 9-I compiles the characteristics of the CP specifications for the case studies used in the following experiments (detailed in sections 5.3.1, 5.3.2.1, 5.3.2.3, 5.3.3). We then generate test frames and test cases

Table 9-I: CP specification of the cases studies used for comparing the proposed criteria

							Choices involved in X																	
Case study	LOC	Parameters	Categories	Choices	Single Choices	Error Choices	X = all constraints	X = 1 constraint	X = 2 constraints	X = 3 constraints	X = 4 constraints	X = 5 constraints	X = 6 constraints	X = 7 constraints	X = 8 constraints	X = 11 constraints	No. of terms in CNF	2-tuple constraints	3-tuple constraints	4-tuple constraints	5-tuple constraints	6-tuple constraints	% of choices constrained	Constraint per choice
Triangle	38	3	9	18	3	3	9	3	0	0	6	0	0	0	0	0	12	9	3	0	0	0	66	1.5
NextDate	179	3	4	18	1	6	6	4	1	1	0	0	0	0	0	0	4	3	1	0	0	0	72	0.84
PackHexChar	211	3	7	32	0	4	26	13	5	1	2	3	1	1	0	0	26	22	0	3	0	1	94	1.0
CS_L1_A	65	5	5	16	1	0	11	7	1	2	1	0	0	0	0	0	11	10	1	0	0	0	75	1.0
CS_L1_C	329	6	6	21	1	0	11	0	6	1	1	0	1	1	0	1	12	2	2	6	2	0	52	1.18
CS_L2_A	1060	11	15	46	0	0	39	21	8	5	0	3	2	0	0	0	27	13	5	7	0	2	85	0.7

(sections 5.4 and 5.5) and obtain measurements (section 5.6) after executing the tests on the product code.

We designed 15 experiments and categorized them in four different categories as shown in Table 9-II.

The first category of experiments generate test frames using different flavours of the Base Choice criterion, i.e., BC, CBC and ECBC. Test frames for BC are generated using Melba in combination with CASA [11]. While accounting for constraints among the choices, the generation of test frames using BC might not exercise each choice because of the removal of infeasible test frames from the test suite. The test frames using CBC and ECBC are generated manually using the algorithms proposed in sections 8.1 and 8.2.

The second, third and fourth categories of experiments deal with the Each Choice, Pairwise and Three-way selection criteria. Since we use two technologies for the generation of test frames (CASA and ACTS) and each technique generates test frames in the presence and in the absence of single annotations, we have four experiments to accommodate these four configurations, which will result in four Each Choice adequate test suites, four Pairwise adequate test suites and four Three-way adequate test suites.

Since there are four test suites for each selection criterion (i.e., 1, 2, 3-way), we select the best test suite for each criterion on the basis of its measurements and compare its result with base choice variants (BC, CBC, ECBC). In addition to mutation score, structural control flow coverage (statement, term and branch, loop coverage) and cost, we will also use structural data flow coverage to measure the quality of the test suite. Data flow coverage is measured as Def Use (DU) associations with the help of Ba-Dua (section 5.6.3). To select the best test suite, we give priority to mutations score, i.e., the

Table 9-II: Experimental Setup for comparison with the proposed criteria

Experiment type	Experiment	Description
Type1: Experiments using BC, CBC and ECBC	Exp 1	BC adequate test suite in the presence of constraints among choices in the CP specification.
	Exp 2	CBC adequate test suite
	Exp 3	ECBC adequate test suite
Type 2: Experiments using Each Choice Criterion	Exp 4	EC adequate test suite in the absence of single annotations using CASA
	Exp 5	EC adequate test suite in the presence of single annotations using CASA
	Exp 6	EC adequate test suite in the absence of single annotation using ACTS
	Exp 7	EC adequate test suite in the presence of single annotations using ACTS
Type 3: Experiments using Pairwise criterion	Exp 8	Pairwise adequate test suite in the absence of single annotations using CASA
	Exp 9	Pairwise adequate test suite in the presence of single annotations using CASA
	Exp 10	Pairwise adequate test suite in the absence of single annotation using ACTS
	Exp 11	Pairwise adequate test suite in the presence of single annotations using ACTS
Type 4: Experiments using 3-way criterion	Exp 12	3-way adequate test suite in the absence of single annotations using CASA
	Exp 13	3-way adequate test suite in the presence of single annotations using CASA
	Exp 14	3-way adequate test suite in the absence of single annotation using ACTS
	Exp 15	3-way adequate test suite in the presence of single annotations using ACTS

test suite which has the highest mutation score is selected. If there are two or more test suites with the same mutation score, we calculate the average of all four types of control flow coverage, i.e., statement, branch and term coverage and select the test suite with highest average. In case of similar control flow coverage, we select the test suite with higher data flow coverage (DU associations). If the data flow coverage is also similar, the suite with the lowest cost (lesser number of test frames) is selected. The quality of the test suite hence selected is compared with BC, CBC and ECBC. Concisely, we are comparing our proposed criterion (BC, CBC and ECBC) with the best of the outcomes of CASA and ACTS, when they are working as combinatorial testing tools (i.e., in the absence of single annotations) or when they are given additional abilities related to the Category Partition technique (i.e., in the presence of single annotations).

9.3 Result of experiments

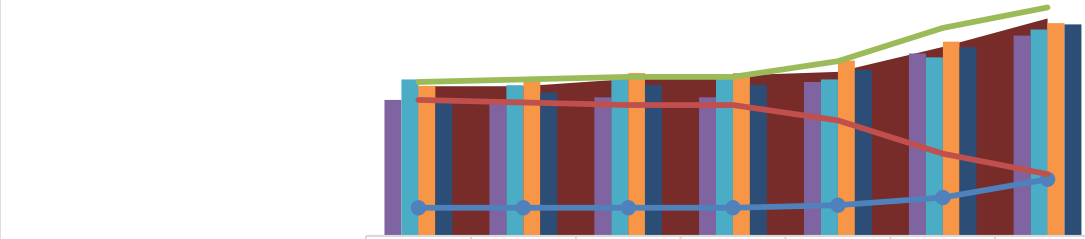
We begin by discussing the experimental results for each case study separately in sections 9.3.1, 9.3.2, 9.3.3, 9.3.4, 9.3.5 and 9.3.6. In each section, i.e., for each case study, we compare the measurements obtained from experiment 1 to experiment 15. Precisely, we compare the results of 1-way, 2-way and 3-way and Base Choice criteria with CBC and ECBC.

9.3.1 The NextDate case study

All results are available in Figure 9-1 (experiments 4, 5, 6 and 7 compared to experiments 1, 2 and 3 in Table 9-II), Figure 9-2 (experiments 8, 9, 10 and 11 vs. 1, 2 and 3) and Figure 9-3 (experiments 12, 13, 14 and 15 vs. 1, 2 and 3), and discussed in sections 9.3.1.1, 9.3.1.2 and 9.3.1.3 respectively. MuClipse creates 113 mutants and Ba-Dua observes a total (maximum) of 72 DU associations to be exercised for the NextDate case study, which has one class and one method. The legend of each figure is self-explanatory; what differs is the criterion, which application is automated with CASA and ACTS (first four bar charts in each figure), that is compared to base choice variants (last three bar charts in each figure).

9.3.1.1 NextDate Each Choice vs BC variants

Figure 9-1: First looking at base choice variants, we observe control flow coverage (statement, branch and term) improves consistently from BC to CBC to ECBC (e.g., statement increases from 60.9% to 69.6% to 80.4%) although with an added number of test frames (e.g., 12 to 15 to 22). The mutation score also improves from 60% to 78% as the number of killed mutants increases from 68 to 89. The reason why CBC is unable to kill as many mutants as ECBC is that CBC does not have a test case with month Dec and



	CASA (-s)	CASA (+s)	ACTS (-s)	ACTS (+s)	BC	CBC	ECBC
Total Data flow coverage (%)	58.3	58.3	61.1	62.5	63.9	73.6	84.7
Mutation Score (%)	53	53	54	54	60	71	78
Statement Coverage (%)	60.9	58.7	60.9	63	60.9	69.6	80.4
Branch Coverage (%)	58.5	61	63.4	63.4	68.3	75.6	82.9
Term Coverage (%)	52.9	55.9	58.8	58.8	64.7	73.5	82.4
Number of frames	11	11	11	11	12	15	22
Mutants alive	53	52	51	51	45	32	24
Mutants Killed	60	61	62	62	68	81	89

Figure 9-1: NextDate—Each Choice vs BC variants

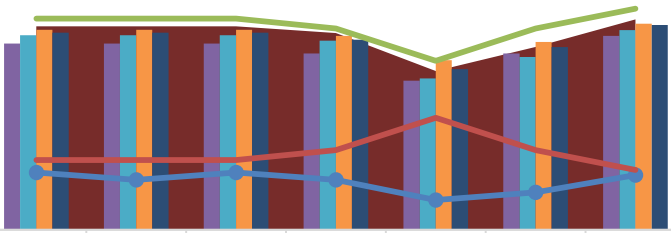
date 30 or 31. The Data flow coverage also improves: from 46 DU pairs (63.9%) covered out of 72 with Base Choice, the number improved to 53 (73.6%) in CBC to 61 (84.7%) for ECBC.

The number of test frames created using all the four experiments involving CASA and ACTS is the same, i.e., 11, showing that single annotations have not helped lowering the cost. We observe that ACTS leads to a marginal increase in mutation score and DU coverage as compared to CASA (54% instead of 53% for mutation score and 62.5% instead of 58.3% for DU coverage). We also observe that single annotations in ACTS led to a rise in statement coverage and DU coverage. There is one additional DU pair coverage when single annotations are used. Since CASA is non-deterministic, a different run may lead to the same result as what we obtain with ACTS. For our experiments, we relied on the results obtained from one execution of CASA. Studying the impact of CASA's non-determinism is out of the scope of this thesis.

Comparing EC with base choice variants, we observe that base choice variants consistently lead to higher mutation score and DU coverage than EC regardless of tooling (ACTS, CASA) or presence of single annotations. Coverage achieved with base choice variants is also higher, with the exception of Each Choice when using ACTS with single annotation: even though EC (with ACTS) covers more statements than BC, it kills fewer mutants, which is due to lower branch coverage. Higher coverage, mutation score and DU coverage is achieved with base choice variants at the expense of increased cost; whether this additional cost is worth it is a difficult question to address as it relates, for instance, to the severity of faults; this issue is out of the scope of this thesis.

9.3.1.2 NextDate Pairwise vs BC variants

We observe (Figure 9-2) that in the absence of single annotations CASA and ACTS perform identically (cost, structural control and data flow coverage and mutation score).



	CASA (-s)	CASA (+s)	ACTS (-s)	ACTS (+s)	BC	CBC	ECBC
Total Data flow coverage (%)	81.9	81.9	81.9	79.2	63.9	73.6	84.7
Mutation Score (%)	75	75	75	71	60	71	78
Statement Coverage (%)	78.3	78.3	78.3	76.1	60.9	69.6	80.4
Branch Coverage (%)	80.5	80.5	80.5	78	68.3	75.6	82.9
Term Coverage (%)	79.4	79.4	79.4	76.5	64.7	73.5	82.4
Number of frames	23	20	23	20	12	15	22
Mutants alive	28	28	28	32	45	32	24
Mutants Killed	85	85	85	81	68	81	89

Figure 9-2: NextDate—Pairwise vs BC variants

However, in the presence of single annotations CASA performs slightly better than ACTS (visit section 7.3.2 for an explanation of this result) with better control flow coverage, DU coverage (CASA covers two additional DU pairs than ACTS) and better mutation score.

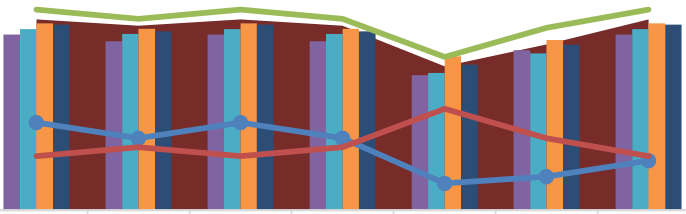
Comparing these results to base choice variants, we observe that pairwise performs better than BC (higher structural control flow coverage and DU coverage and mutation score) though at a higher cost (20 or 23 against 12): pairs of choices trigger various input conditions which trigger specific branches and DU pairs. The best pairwise test suite, i.e., CASA(+s), performs better than CBC by killing 4% more mutants and covering six more DU pairs but with an additional cost of five test cases. The better mutation score and DU coverage is resulting from a test case with Dec 31st and 28th Feb on a non-leap year, that is missing in CBC. Nevertheless, looking at sets of test frames, we observe few frames in BC and CBC test suites that do not appear in PW adequate test suites.

On the contrary, ECBC performs better than PW, regardless of PW configurations, with a better mutation score, structural control flow coverage and DU coverage at a comparable cost. The difference in mutation score is due to the absence from PW test suites, including when using CASA, of a test case with leap year and 28th Feb. This 3-way combination is present in CBC and ECBC. ECBC performs better that PW in terms of statement, branch and term coverage with some choice combination, i.e., test frames in the ECBC test suite that are not found in PW test suites. Some mutants are specifically killed by PW (because of pairwise combinations) and some mutants are specifically killed by ECBC (because of some 3-way combinations). However, all the mutants that are killed by PW are also killed by ECBC. Further ECBC covers two more DU pairs as

compared to the test suite created using CASA in the presence of single annotations, i.e., ECBC covers 61 DU associations as compared to 59 for CASA PW out of the total 72.

9.3.1.3 NextDate 3-way vs BC variants

Not using single annotations leads to better results (statement, branch and term coverage, mutation score and data flow coverage) regardless of the tool (CASA, ACTS), though at a higher cost, and CASA and ACTS produce identical results for the same configuration (e.g., using single annotations): Figure 9-3. If we compare the results of ACTS (-s), which is the best and is similar to CASA (-s) amongst all the 3-way combination test suites with base choice variants, we observe that 3-way performs better than BC and CBC in terms of mutation score, DU and statement, branch and term coverage although with almost double the cost. But comparing the results with ECBC we observe that ECBC scores equivalent to the 3-way criterion: slightly higher or equal



	CASA (-s)	CASA (+s)	ACTS (-s)	ACTS (+s)	BC	CBC	ECBC
Total Data flow coverage (%)	84.7	81.9	84.7	81.9	63.9	73.6	84.7
Mutation Score (%)	78	75	78	75	60	71	78
Statement Coverage (%)	80.4	78.3	80.4	78.3	60.9	69.6	80.4
Branch Coverage (%)	82.9	80.5	82.9	80.5	68.3	75.6	82.9
Term Coverage (%)	82.4	79.4	82.4	79.4	64.7	73.5	82.4
Number of frames	39	32	39	32	12	15	22
Mutants alive	24	28	24	28	45	32	24
Mutants Killed	89	85	89	85	68	81	89

Figure 9-3: NextDate—3-way vs BC variants

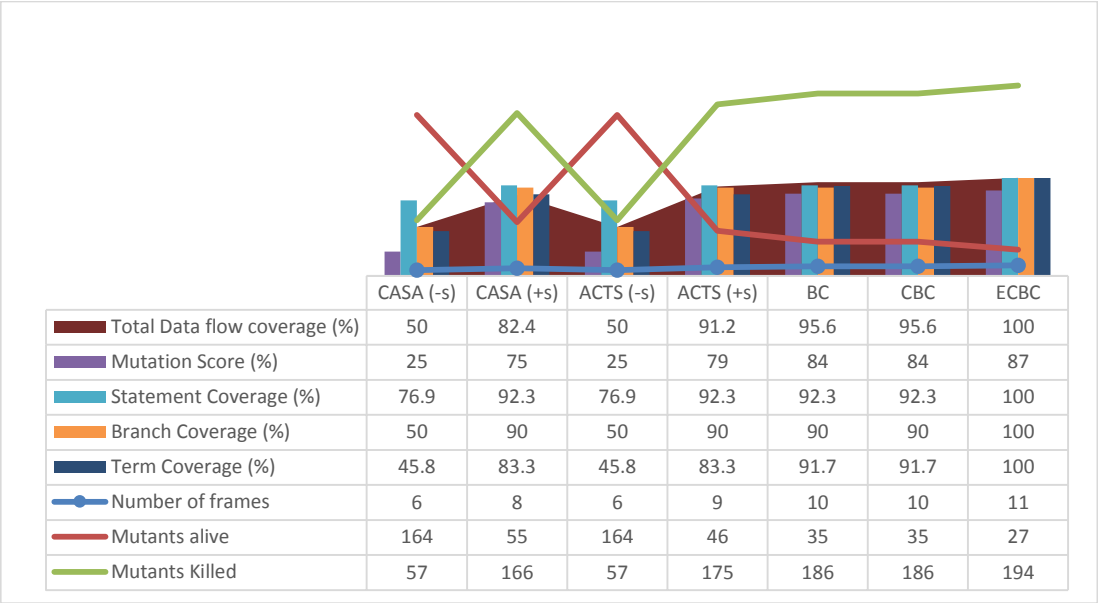
mutation score (78%) and DU coverage (84.7%), slightly higher or equal statement coverage; ECBC is significantly cheaper than 3-way (22 tests as opposed to 32 or 39 tests). When mutation score is identical between 3-way and ECBC (i.e., 3-way omitting single choices, regardless of tool), the two criteria kill exactly the same mutants and cover exactly the same DU associations. When ECBC outperforms 3-way (i.e., 3-way with single choices, regardless of tool), ECBC kills all the mutants also killed by 3-way plus four additional ones. Additionally, ECBC covers all the DU associations covered by 3-way plus two more because ECBC has a combination of Feb and day less than 28 which is missing in 3-way. The combination of leap year and 28th Feb, which we said is missing in PW, is present with the 3-way criterion test suites.

9.3.2 Triangle Case Study

All results are available in Figure 9-4 (experiments 4, 5, 6 and 7 compared to experiments 1, 2 and 3 in Table 9-II), Figure 9-5 (experiments 8, 9, 10 and 11 vs. 1, 2 and 3) and Figure 9-6 (experiments 12, 13, 14 and 15 vs. 1, 2 and 3), and discussed in sections 9.3.2.1, 9.3.2.2 and 9.3.2.3 respectively. MuClipse creates 221 mutants and Ba-Dua observes 68 DU associations for the Triangle case study that has one class and two methods.

9.3.2.1 Triangle Each Choice vs BC variants

Figure 9-4: Results for BC and CBC are identical. This is because the test suite generated using BC covers each choice at least once; this is due to the fact that constraints on choices do not lead to the generation of infeasible test frames; as a result CBC did not add any more test frame. ECBC introduced an additional test frame to CBC



	CASA (-s)	CASA (+s)	ACTS (-s)	ACTS (+s)	BC	CBC	ECBC
Total Data flow coverage (%)	50	82.4	50	91.2	95.6	95.6	100
Mutation Score (%)	25	75	25	79	84	84	87
Statement Coverage (%)	76.9	92.3	76.9	92.3	92.3	92.3	100
Branch Coverage (%)	50	90	50	90	90	90	100
Term Coverage (%)	45.8	83.3	45.8	83.3	91.7	91.7	100
Number of frames	6	8	6	9	10	10	11
Mutants alive	164	55	164	46	35	35	27
Mutants Killed	57	166	57	175	186	186	194

Figure 9-4: Triangle Each Choice vs BC variants

leading to improved mutation score (87%), control flow coverage (100%) and DU coverage (100%). The additional test frame corresponds to a test case for an equilateral triangle, which was missing in the test suites for BC and CBC.

The four Each Choice adequate test suites resulting from experiments 4, 5, 6 and 7 lead to lower mutation score, statement, branch, term and DU coverage as compared to Base Choice variants. In these cases, using single annotations resulted in better quality than not using them. Choices annotated as single are the ones corresponding to invalid triangle inequalities. Using the annotations ensures that invalid triangle inequalities are exercised separately from other choice combinations; consequently, the remaining choices are better involved in interesting combinations, leading to higher control flow coverage and DU coverage and mutation score.

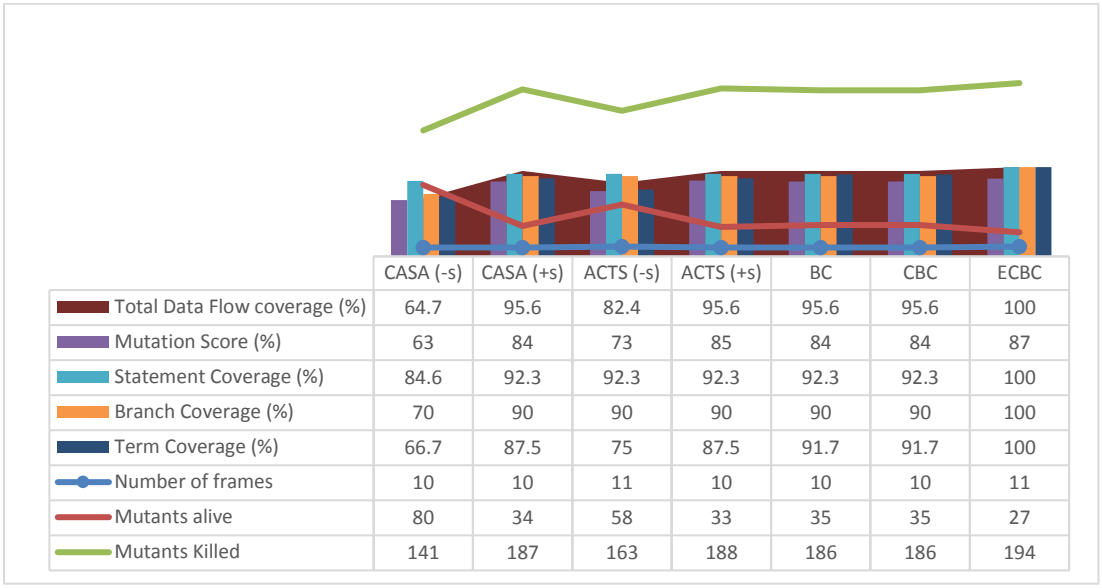
When those annotations are not used, the choices corresponding to invalid triangle inequalities are combined with choices for valid inputs which, because of the invalid inequality input, do not have a chance to exercise functionally interesting pieces of code.

ACTS with single annotations killed more mutants than CASA with single annotations with an additional test case. This made CASA (+s) and ACTS (+s) as candidates for comparison with Base Choice and its variants.

Base Choice variants perform better than Each Choice, regardless of configuration (tool, presence of single annotations), with little increased cost. Comparing with ACTS (+s) shows that improved mutation score is due to improved term coverage.

9.3.2.2 Triangle Pairwise vs BC variants

Comparing results of experiments 8, 9, 10 and 11 we again observe that using Single annotations pays off (Figure 9-5). The reason for this behaviour is explained in section 7.4.2. This is clearer for CASA than for ACTS: for ACTS, mutation score increases when using single annotations because of increased Term coverage. ACTS (+s) leads to equal DU coverage but higher mutation score than BC and CBC, at the same cost, even-though BC and CBC have identical statement and branch coverage and higher term coverage



	CASA (-s)	CASA (+s)	ACTS (-s)	ACTS (+s)	BC	CBC	ECBC
Total Data Flow coverage (%)	64.7	95.6	82.4	95.6	95.6	95.6	100
Mutation Score (%)	63	84	73	85	84	84	87
Statement Coverage (%)	84.6	92.3	92.3	92.3	92.3	92.3	100
Branch Coverage (%)	70	90	90	90	90	90	100
Term Coverage (%)	66.7	87.5	75	87.5	91.7	91.7	100
Number of frames	10	10	11	10	10	10	11
Mutants alive	80	34	58	33	35	35	27
Mutants Killed	141	187	163	188	186	186	194

Figure 9-5: Triangle Pairwise vs BC variants

than ACTS but ACTS (+s) kills two more mutants than BC/CBC.

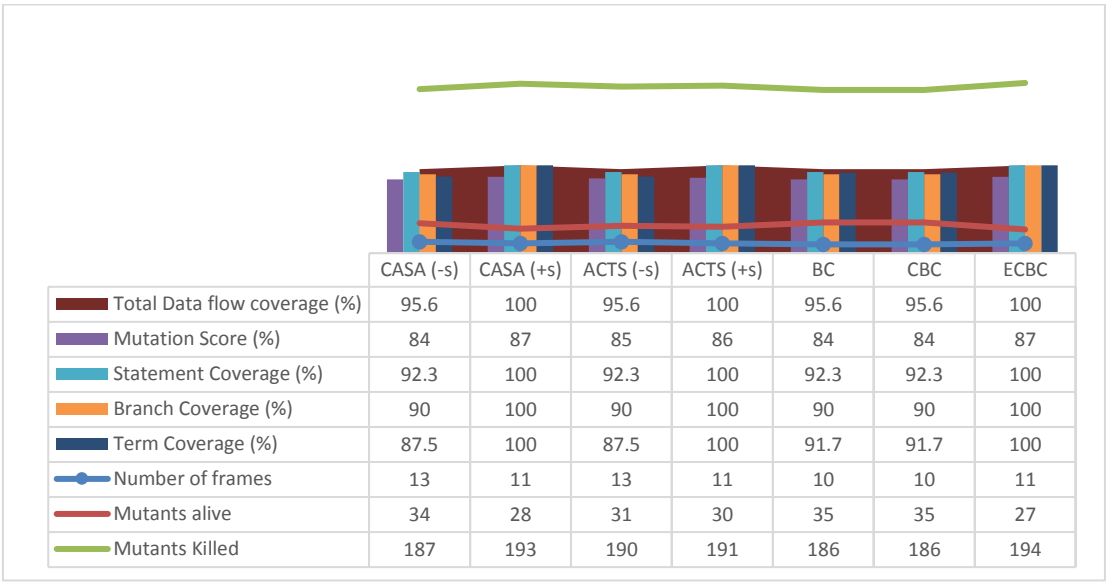
In addition to the fact that both the test suites kill non-inclusive set of mutants, we observe that BC and CBC lacks the test case for equilateral triangle which is present in the test suite for ACTS (+s), therefore all the mutants that are killed by ACTS (+s) and not by BC/CBC are related to equilateral triangle inputs. We observe that there is one mutant that is killed by ACTS (+s) but is not killed by CBC and ECBC. This mutant corresponds to test input selection for an isosceles triangle. The test input given to ACTS (+s) for an isosceles triangle is (4, 4, 3) whereas that given to CBC and ECBC is (3, 3, 5). When the code (b==c) is replaced by the mutant (b≥c), the input (4, 4, 3) kills the mutant whereas the input (3, 3, 5) cannot. Making a similar selection gives similar results. On the other hand, we also observe that all the mutants that are not killed by ACTS (+s) are killed by the BC and CBC. In terms of DU coverage, we observe that although both ACTS (+s) and BC/CBC test suites have the same DU coverage, a small number of pairs covered by both the test suites are non-overlapping. This is essentially because they have different term coverage resulting from different test cases. ACTS (+s) performs slightly worse than ECBC as the test suite it creates does not contain a test case for a scalene triangle: such a test requires a 3-way combination rather than a pairwise combination. Both ACTS and CASA cannot generate this test frame. ECBC does. It is for the same reason that ACTS (+s) does not cover three DU pairs which are covered by the ECBC test suite. Comparing the mutants killed by ECBC and ACTS (+s) we again observe that all the mutants that are not killed by ACTS (+s) are killed by ECBC.

9.3.2.3 Triangle 3-way vs BC variants

Similar to previous experiments, we observe that using Single annotations pays off both in terms of control flow coverage (all coverage criteria reach 100%), DU coverage (100%) and mutation score (e.g., from 84% to 87% with CASA):

Figure 9-6. BC and CBC are very close to 3-way, in terms of cost, mutation score and DU coverage (difference of three DU pairs), regardless of tool and presence of single annotations. ECBC outperforms all four test suites of 3-way, especially CASA and ACTS with single annotations, even though all test suites achieve 100% control flow coverage for all three coverage criteria.

ECBC has the same DU coverage, as it covers all 68 DU pairs similar to ACTS (+s) and CASA (+s). We observe that ECBC kills one more mutant than the mutants killed by CASA (+s), which is the best quality test suite among 3-way test suites. From a qualitative analysis we attribute this to the selection of test inputs. The ECBC test suite



	CASA (-s)	CASA (+s)	ACTS (-s)	ACTS (+s)	BC	CBC	ECBC
Total Data flow coverage (%)	95.6	100	95.6	100	95.6	95.6	100
Mutation Score (%)	84	87	85	86	84	84	87
Statement Coverage (%)	92.3	100	92.3	100	92.3	92.3	100
Branch Coverage (%)	90	100	90	100	90	90	100
Term Coverage (%)	87.5	100	87.5	100	91.7	91.7	100
Number of frames	13	11	13	11	10	10	11
Mutants alive	34	28	31	30	35	35	27
Mutants Killed	187	193	190	191	186	186	194

Figure 9-6: Triangle 3-way vs BC variants

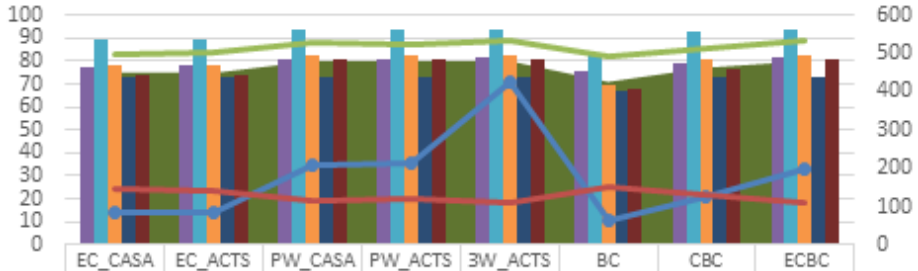
kills this specific mutant when the test inputs is (26, 5, 6): this is an invalid triangle which satisfies one triangle inequality (a>b+c); But by replacing (b+c) in the code with mutant (b*c) the inequality is not satisfied and the triangle becomes a valid triangle resulting in the killing of the mutant. In case of CASA (+s), the random values we give to the invalid triangle is (30, 3, 3) and these values cannot kill the mutant (b*c) since the inequality is still satisfied. Hence the mutant survives. Therefore, for the triangle case study the killing of the mutants is also affected by the test input selection. For instance, selecting the test inputs for an isosceles triangle impacts how the mutants are killed.

9.3.3 PackHexChar Case Study

All the results are compiled in Figure 9-7. We have only one version of CP specifications of PackHexChar because there are no Single annotations for choices in this case study. Similar to other case studies we used CASA and ACTS for the generation of test frames and we generated Each Choice, Pairwise and 3-Way adequate test suites. While generating 3-way adequate test suites, we were able to obtain the results with ACTS but the execution with CASA kept executing for more than 24 hours and we decided to terminate the execution. So, we are reporting on resulting from 3-way test suites obtained from ACTS only. The results in Figure 9-7 corresponds to experiment 1, 2, 3 (for BC variants) together with experiment 4 and 6 (Each choice), experiment 8 and 10 (Pairwise) and experiment 14 (3-way ACTS) in Table 9-II. The results are discussed in detail in sections 9.3.3.1, 9.3.3.2 and 9.3.3.3. MuClipse created 643 mutants for this case study and Ba-Dua observed 307 DU associations for the PackHexChar case study that has one class and nine methods. Figure 9-7 shows the percentage of total DU coverage for the whole class.

9.3.3.1 PackHexChar Each Choice vs BC variants

Both the Each Choice adequate test suites obtained from ACTS and CASA have comparable results (Figure 9-7). They have similar cost and similar control flow coverage. There is however a difference of about 1% in mutation score and data flow coverage. CASA has 1% lower mutation score but covers one additional DU association. An additional DU association is resulting from a specific test frame combination that is present in the CASA test suite but is missing in the ACTS test suite. This combination resulted in an RLEN substring where the first character is a non-hexadecimal. Although this resulted in a bit higher DU coverage, its effect is not very significant in killing mutants. ACTS however killed more mutants because it has a test input selection where hexadecimal character F exists in the RLEN substring. CASA lacks such an input selection.



	EC_CASA	EC_ACTS	PW_CASA	PW_ACTS	3W_ACTS	BC	CBC	ECBC
Data flow coverage (%)	75.2	74.9	80.1	80.1	80.1	71	77.2	80.1
Mutation Score (%)	77	78	81	81	82	76	79	82
Statement Coverage (%)	89.7	89.7	94.1	94.1	94.1	82.4	92.6	94.1
Branch Coverage (%)	78.3	78.3	82.6	82.6	82.6	69.6	80.4	82.6
Loop Coverage (%)	73.3	73.3	73.3	73.3	73.3	66.7	73.3	73.3
Term Coverage (%)	73.8	73.8	81	81	81	67.9	76.2	81
Number of frames	14	14	35	36	71	11	21	33
Mutants alive	147	141	116	118	110	152	129	111
Mutants Killed	496	502	527	525	533	491	514	532

Figure 9-7: PackHexChar Each Choice vs Pairwise vs 3-way vs BC variants

Looking at BC variants, we observe that the control flow coverage, mutation score and DU coverage improves consistently from BC to CBC to ECBC. Because of the complexity of constraints (Table 9-I), the Base Choice test suite is not Each Choice adequate, thereby resulting in lower measurements than both CASA and ACTS. The CBC criterion resulted in an Each Choice adequate test suite while satisfying the constraints which resulted in better mutation score, DU coverage along with higher control flow coverage. The CBC test suite kills 12 to 18 more mutants than both the Each Choice test suites and covers 6 to 7 more DU pairs which are otherwise missed by Each Choice test suites. All these improvements however resulted from an additional cost of seven test cases.

ECBC test suite has control flow coverage, mutation score and DU coverage greater than the BC and CBC test suites and both the Each Choice test suites at a cost which is greater than all the above mentioned test suites. Therefore, clearly the BC variants (CBC and ECBC) performed better than Each Choice (ACTS/CASA) although at an additional cost.

9.3.3.2 PackHexChar Pairwise vs BC variants

Comparing the results of experiments 8 and 10 we observe that ACTS is slightly costlier than CASA with one additional test frame. The DU coverage and control flow coverage for both the test suites is similar. CASA however killed two more mutants than ACTS. On deeper mutation analysis, we observed that ACTS killed one mutant that is not killed by CASA or any of the BC variants. This mutant is killed only if the number of hexadecimal characters in the RLEN first characters is even but not 2 and there should be no non-hexadecimal character in the RLEN first characters. In other words, this specific

test frame results from a 3-way combination that is only present in ACTS and is absent in other BC variants. Since CASA is non-deterministic, other executions of CASA might generate this combination but we are relying on the first execution. CASA killed three more mutants that survived ACTS and the reason for their survival is a specific test input. For instance, when a test case specification required odd hexadecimal characters, we randomly opted for five characters for ACTS and three for CASA. These sorts of selections led to the survival of the mutants. Given different test input ACTS could also kill the three mutants that are killed by CASA.

Comparing the Pairwise adequate test suites with Base Choice variants we observe that the PW test suite performs better than the Base Choice and CBC test suites with a larger difference with the former but lesser difference with the latter. Comparing CASA with CBC, we observe that with about 40% more cost CASA kills just 2% more mutants and covers 3% more DU associations.

Comparing ECBC with PW test suites we observe that ECBC performs better than both the PW suites by killing 1% more mutants at a lesser cost (Figure 9-7). ECBC killed five more mutants than CASA with two lesser test frames and killed seven more mutants than ACTS with three lesser test frames. We further analyzed the mutants that are killed by ECBC but survived PW test frames. We observe that the difference is because of the test input selection. Although we ensure that all the test suites have input strings with hexadecimal characters from A to F, but we did not ensure the same in each RLEN substring. The presence of all the characters would also depend on the fact that the size of the RLEN substring should be greater than or equal to six in combination with other choices as per the constraints. The ECBC test suite has test inputs in RLEN substring that

range from A to F hexadecimal characters whereas both PW test suite have A to F characters in the input string but the RLEN substring closely missed F. It is because of this difference in test inputs that the ECBC test suite killed five more mutants than the CASA and ACTS test suites. Nevertheless, it is essential to know that ECBC obtained these results at a cost that is less than CASA and ACTS.

9.3.3.3 PackHexChar 3-way vs BC variants

As mentioned earlier, we have results of 3-way combinations only from ACTS. We did not get results from CASA, which exceeded 24 hours of execution without any outcome, so we terminated the execution. ACTS outperformed Base Choice and CBC in terms of mutation score, DU coverage and control flow coverage but at a cost that is tremendously higher. ACTS has 71 test frames compared to 11 for Base Choice and 21 for CBC (Figure 9-7)

We have interesting conclusions from the comparison of ACTS with ECBC. We observe that ACTS performs similarly to ECBC but with 71 test frames as compared to 33 test frames for ECBC. Both the ECBC test suite and ACTS test suite cover an equal number of DU associations (80.1%) and achieve equal control flow coverage (statement 94.1%, branch 82.6%, loop 73.3% and term 81%). There is however a difference of one mutant out of 643 which is killed by the ACTS test suite and which survived the ECBC test suite leading to a mutation score of 82%. This mutant, which is also killed by the ACTS Pairwise test suite (discussed in section 9.3.3.2), requires a specific three-way combination that exists in ACTS three-way but not in ECBC. Therefore, three-way by ACTS obtained all the comparable results at a cost that is 54% higher than ECBC.

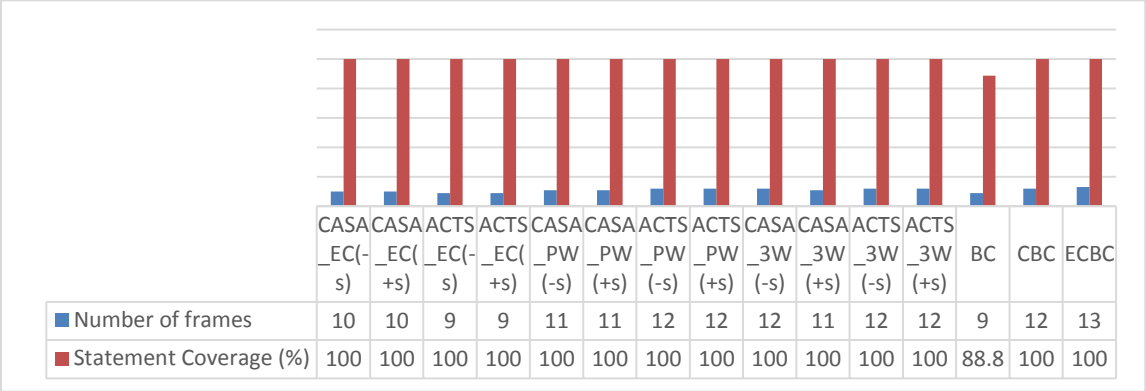


Figure 9-8: CS_L1_A Each Choice vs Pairwise vs 3-way vs BC variants

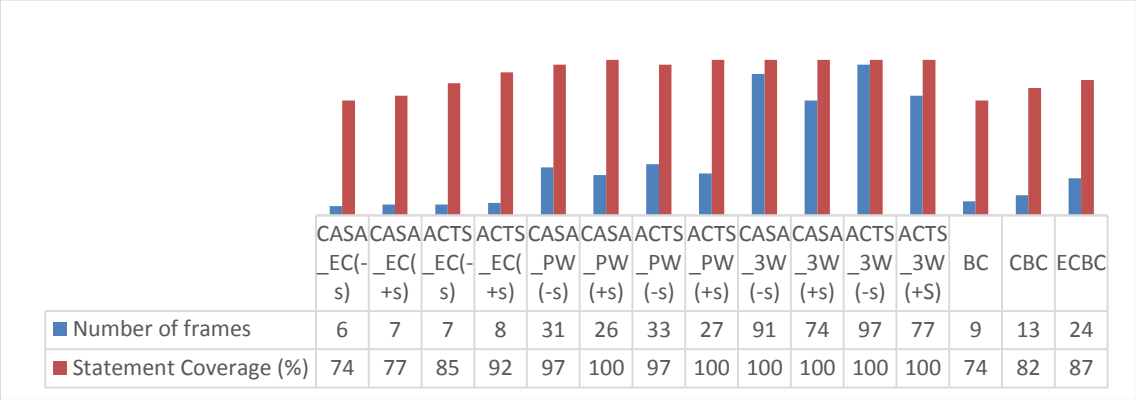
9.3.4 Industrial case study CS_L1_A

The results from all the experiments for industrial case studies are compiled as one figure.

In Figure 9-8 the results of the experiments in the presence and absence of single annotations for both CASA and ACTS are similar. All the selection criteria except Base Choice result in 100% statement coverage because the Base Choice in the presence of constraints does not cover every permitted choice. Every choice is covered in CBC and ECBC, resulting in better coverage than Base Choice. The cost of the test suite is also comparable and no major difference is observed.

9.3.5 Industrial case study CS_L1_C

The quality of the test suite in the presence of Single annotations is better than in their absence for both combinatorial testing tools and for all selection criteria (Figure 9-9): e.g., 97% and 100% statement coverage for ACTS (-s) and ACTS (+s) respectively, for PW. Identifying single choices restricts their presence in other combinations, resulting in more opportunities for other choices to be involved in interesting combinations. The cost of test suites generated by CASA is always less than that of ACTS test suites,



	CASA_EC(-s)	CASA_EC(+s)	ACTS_EC(-s)	ACTS_EC(+s)	CASA_PW(-s)	CASA_PW(+s)	ACTS_PW(-s)	ACTS_PW(+s)	CASA_3W(-s)	CASA_3W(+s)	ACTS_3W(-s)	ACTS_3W(+S)	BC	CBC	ECBC
Number of frames	6	7	7	8	31	26	33	27	91	74	97	77	9	13	24
Statement Coverage (%)	74	77	85	92	97	100	97	100	100	100	100	100	74	82	87

Figure 9-9: CS_L1_C Each Choice vs Pairwise vs 3-way vs BC variants

regardless of configuration and criterion. We observe that CBC performs better than EC with CASA and ECBC performs better than EC in the absence of single annotation, i.e., when a regular application of a combinatorial testing tool is taken into consideration, for both CASA and ACTS.

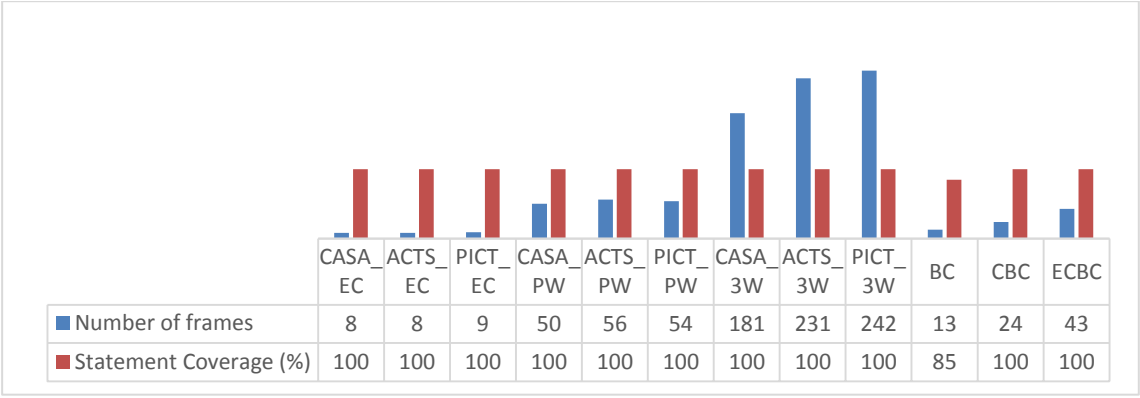
A Pairwise and 3-way adequate test suite, in the presence or absence of single annotations performs better than Base Choice and its variants for this case study. The reduced coverage of CBC/ECBC as compared to PW is attributed to the fact that there can be more than one suitable candidate to be a base choice for each category in this CP specification; choosing one such base choice and applying BC/CBC/ECBC privileges this particular choice whereas PW does not and considers all the choices in the same way. As a result, selecting a particular choice as base choice results in fewer combinations involving this choice and other choices, resulting in lower coverage. Grindal et. al. [7] explain this phenomenon by stating that if there are too many candidates for base choice then PW is a better criterion to choose than BC. However, a qualitative analysis of the ECBC test suite shows, similarly to previous case studies, that its test frames show traits of not only pairwise but also 3-way and 4-way selection criteria. We observed that a

reason Pairwise test suites obtained from CASA (-s) and ACTS (-s) cannot reach 100% structural coverage was a specific missing combination of choices. +S with ACTS and CASA result in this combination and therefore 100% code coverage and so does 3-way. We confirm the presence of this combination in the test suite generated using CBC and ECBC. Therefore, we conclude that the test suite generated using CBC and ECBC shows the presence of some important combinations of choices that are otherwise not possible in one single n-way selection criteria but can be obtained from a variable strength selection criterion.

9.3.6 Industrial case study CS_L2_A

For case study CS_L2_A we used three combinatorial testing tools; CASA, ACTS, PICT selected from our study [16] instead of the previous two tools. All three tools differ in the technique they use for the generation of test frames. There are no Single annotations for this case study, therefore we have results from 12 experiments which includes three different tools: Figure 9-10.

The cheapest of all the selection criteria, i.e., Each Choice, results in 100% statement



	CASA_EC	ACTS_EC	PICT_EC	CASA_PW	ACTS_PW	PICT_PW	CASA_3W	ACTS_3W	PICT_3W	BC	CBC	ECBC
Number of frames	8	8	9	50	56	54	181	231	242	13	24	43
Statement Coverage (%)	100	100	100	100	100	100	100	100	100	85	100	100

Figure 9-10: CS_L2_A Each Choice vs Pairwise vs 3-way vs BC variants

coverage for this case study, so all subsequent executions, for other selection criteria will help us study the cost of the test suite and compare it with BC and its variants. Since this case study is highly complex (see CP specification in Appendix C.11), the generation of test frames on the basis of Base Choice in the presence of constraints results in a lot of invalid test frames. These invalid frames are removed resulting in low code coverage since the resulting test suite is then not Each Choice adequate. However, with CBC and ECBC, since both criteria subsume Each Choice, the code coverage improved to 100%, although with almost double the cost as compared to Base Choice.

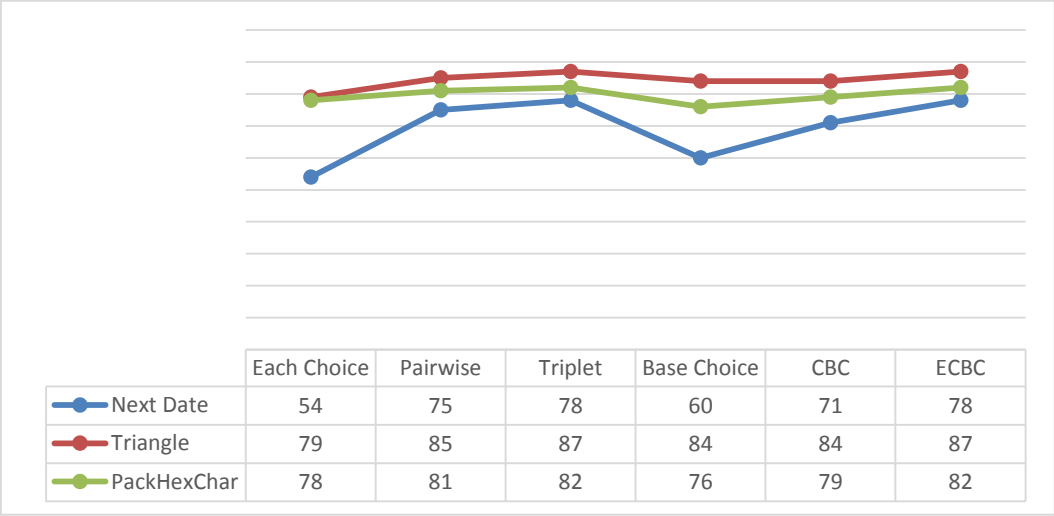
Comparing BC and its variants with Each Choice we observe that CBC and ECBC are more costly than Each Choice. However, comparing CBC and ECBC with Pairwise and 3-way, we observe that the cost of test frames generated using CBC and ECBC is significantly lower than that of Pairwise and 3-way.

If we compare the technologies for the generation of test frames, we observe that CASA is cheaper than ACTS and PICT for all three criteria. This can be attributed to the fact that CASA uses a metaheuristic technique for the generation of test frames whereas the other two use a greedy algorithm. On the other hand, CASA takes longer to generate test frames.

9.4 Discussion on research question

This section answers the research questions identified in section 9.1. We begin by selecting the best Each Choice, Pairwise and 3-way adequate test suites out of the four test suites (i.e., ACTS (-s) ACTS (+s), CASA (-s), CASA (+s)) for each case study as already discussed in the previous sections. The selected suite and BC is compared with CBC and ECBC. Finally, on the basis of results from all the case studies we draw our

final conclusions and answer the research questions. Figure 9-11, Figure 9-12, Figure 9-13, Figure 9-14 compile the results for mutation score, data flow coverage, number of test frames and control flow coverage obtained from the best test suite for all the case studies and the results are compared with CBC and ECBC. Figure 9-11 has results for mutation score and Figure 9-12 has the results for data flow coverage for only academic case studies, i.e., Triangle, NextDate and PackHexChar, since we could not perform mutation and data flow analyses on the proprietary software. Figure 9-13 has results for cost, i.e., number of test frames, for all the case studies. Figure 9-14 shows the results for control flow coverage for all the case studies. Recall that for academic case studies, we have three types of code coverage (statement, branch, term and loop coverage) and for industrial case studies, we have measurement for only statement coverage. For the purpose of representation on the graph and comparisons, we calculated the average of all the three types of coverage measures for academic case studies and represented them along with statement coverage obtained for industrial case studies.



	Each Choice	Pairwise	Triplet	Base Choice	CBC	ECBC
Next Date	54	75	78	60	71	78
Triangle	79	85	87	84	84	87
PackHexChar	78	81	82	76	79	82

Figure 9-11: Comparing mutation score of CBC and ECBC with other criteria

9.4.1 RQ1: How do CBC and ECBC perform as compared to BC

We observe (Figure 9-11, Figure 9-12, Figure 9-13, Figure 9-14) specifically for the Triangle case study, that CBC performs at least as well as BC. In the CP specification where base choices do not have constraints involving each other and a Base Choice adequate test suite is therefore Each Choice adequate (recall from the introduction that BC does not necessarily subsume EC), CBC yields results similar to BC. However, when base choices have constraints involving each other and their combinations as per the Base Choice criterion lead to invalid test frames thereby resulting in the removal of those frames from the test suite, CBC performs better than BC. ECBC subsumes CBC, and we observe that ECBC performs better than CBC.

Cost increases constantly from BC to CBC to ECBC: Figure 9-13. If base choices are not constrained and applying BC results in feasible test frames, then BC and CBC have the same cost: Triangle case study. If applying BC results in discarding choice combinations that are found infeasible, CBC costs more than BC (e.g., NextDate), but also has better effectiveness at finding faults than BC. In case of complex constraints, ECBC costs much more than CBC (e.g., CS_L2_A) whereas without complex constraints, ECBC is slightly more expensive than CBC (e.g., Triangle).

We observe in Figure 9-12 that the Data flow coverage of CBC is the same as BC when the BC adequate test suite is Each Choice adequate; otherwise, CBC covers more DU pairs than BC. Further, ECBC leads to greater data flow coverage than BC and CBC.

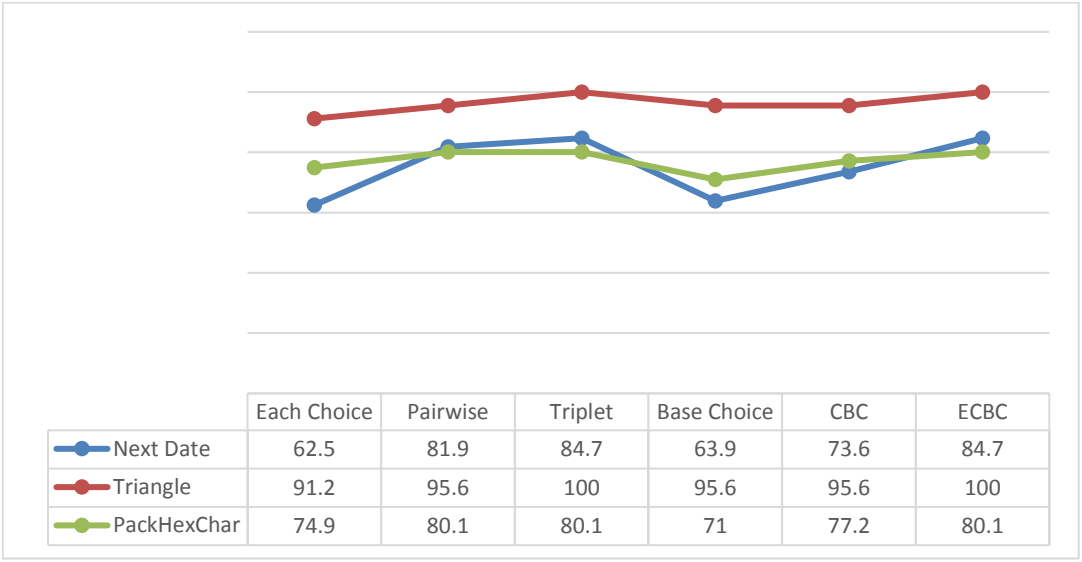
We observe (Figure 9-14) that control flow coverage constantly improves from BC to CBC to ECBC, except for Triangle where the BC adequate test suite is also EC

adequate. We see coverage improves to 100% from BC to ECBC for two case studies and a considerable improvement in the remaining two.

We conclude that the test suite generated using CBC always performs better in terms of mutation score, data and control flow coverage than BC with a higher cost when the test suite generated using BC is not Each Choice adequate (because of constraints). In case the BC adequate test suite is also EC adequate, CBC performs similarly to BC. A test suite generated using ECBC always performs better than those generated using CBC and BC with higher mutation score, data and control flow coverage but with higher cost. As mentioned previously, whether the increased performance is worth the increased cost is out of the scope of this thesis.

9.4.2 RQ2: How do CBC and ECBC perform as compared to EC

We compared the quality of CBC and ECBC adequate test suites with the best EC adequate test suite among the four suites generated by CASA and ACTS in the presence



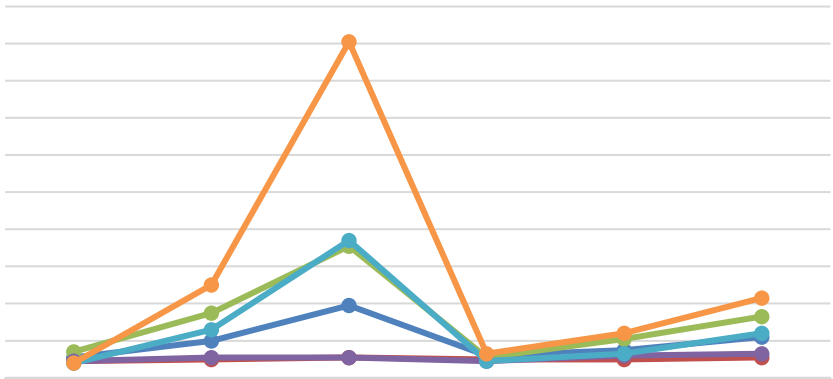
	Each Choice	Pairwise	Triplet	Base Choice	CBC	ECBC
Next Date	62.5	81.9	84.7	63.9	73.6	84.7
Triangle	91.2	95.6	100	95.6	95.6	100
PackHexChar	74.9	80.1	80.1	71	77.2	80.1

Figure 9-12: Comparing Data flow coverage of CBC and ECBC with other criteria

and in the absence of single annotations, for each case study. In order to select the best test suite, similar to section 9.3, we gave highest priority to mutation score followed by control flow coverage, data flow coverage and cost.

The mutation score obtained with CBC is always better than that obtained with EC for the three academic case studies for which we measure mutation score (Figure 9-11). Similarly, ECBC kills more mutants than BC and CBC and performs better. CBC and ECBC always cover more DU pairs than the best Each Choice test suite (Figure 9-12). For the Triangle case study, the DU coverage improved to 100% for the ECBC test suite.

CBC and ECBC are always more costly than EC (Figure 9-13). We observe this result for all the case studies. The increased cost for CBC and ECBC is the result of deviation from base choices while satisfying the constraints. CBC and ECBC perform

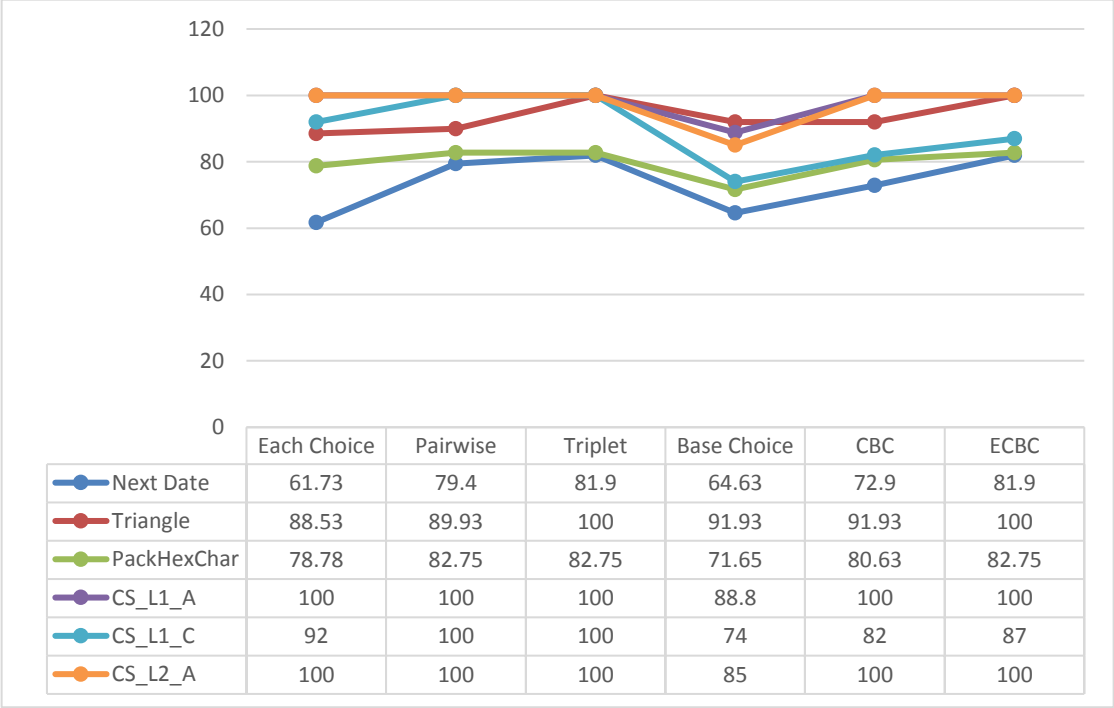


	Each Choice	Pairwise	Triplet	Base Choice	CBC	ECBC
Next Date	11	20	39	12	15	22
Triangle	9	10	11	10	10	11
PackHexChar	14	35	71	11	21	33
CS_L1_A	9	11	11	9	12	13
CS_L1_C	8	26	74	9	13	24
CS_L2_A	8	50	181	13	24	43

Figure 9-13: Comparing cost of CBC and ECBC with other criteria

better than or equally to EC in terms of control flow coverage in five out of six case studies (Figure 9-14). For industrial case study CS_L1_C, CBC and ECBC cover less than EC. As explained in section 9.3.5, CS_L1_C is a better candidate for a combinatorial criterion than the Base Choice criterion (and variations) because more than one choice in each category is suitable for being a base choice. For such a CP specification, BC is not always useful.

We conclude that CBC and ECBC provide better test suite quality than EC, in terms of coverage and fault detection, at the expense of higher cost. One should however be careful since the performance of Base Choice, and its variations, depends on whether one can identify one unique base choice for each category.



	Each Choice	Pairwise	Triplet	Base Choice	CBC	ECBC
Next Date	61.73	79.4	81.9	64.63	72.9	81.9
Triangle	88.53	89.93	100	91.93	91.93	100
PackHexChar	78.78	82.75	82.75	71.65	80.63	82.75
CS_L1_A	100	100	100	88.8	100	100
CS_L1_C	92	100	100	74	82	87
CS_L2_A	100	100	100	85	100	100

Figure 9-14: Comparing control flow coverage of CBC and ECBC with other criteria

9.4.3 RQ3: How do CBC and ECBC perform as compared to PW?

We observe that CBC performs slightly worse than PW, whereas ECBC performs better than PW for the academic case studies. We conclude these results from mutation analysis (Figure 9-11) and data flow coverage (Figure 9-12)

The cost of CBC is in general significantly lower than PW (Figure 9-13), whereas the cost of ECBC is usually on par with that of PW unless the CP specification has complex constraints in which case ECBC is less expensive than PW, i.e., PackHexChar, CS_L1_C and CS_L2_A. Recall that for case studies where ECBC is more expensive than PW, we are considering the execution of PW in the presence of single annotations which are comparatively less expensive than the executions in their absence, in which case the cost would be at par with PW.

We observe (Figure 9-14) that for CS_L1_C case study, CBC and ECBC cover less than PW. We have already explained this difference in coverage in section 9.3.5. For NextDate and PackHexChar, CBC covers less than PW because of insufficient combinations among choices. For the other case studies, ECBC has coverage equal or greater than PW thereby showing that ECBC always performs better or equal to PW.

We conclude that the quality of a test suite created using CBC is equal to or lower than that created using PW at a smaller cost, unlike a test suite created using ECBC, which has a greater or equal quality, at comparable cost. A test suite generated using ECBC has greater mutation score, greater data flow coverage and greater or equal control flow coverage, and almost comparable cost, than a test suite generated using PW.

9.4.4 RQ4: How do CBC and ECBC perform as compared to 3-way?

CBC finds fewer faults and covers fewer DU associations than 3-way while ECBC finds exactly equal faults and covers exactly similar DU associations as 3-way(Figure 9-11, Figure 9-12), suggesting that ECBC has a behavior similar to a 3-way selection criterion.

The cost of CBC and ECBC is always lower than 3-way with one exception, when the CP specification is simple leading to one or two more test frames (Figure 9-13). For instance, for the NextDate case study, ECBC is 44% cheaper than 3-way and for PackHexChar, ECBC is 54% cheaper while producing the same mutation score, data and control flow coverage. For the industrial case studies, ECBC is much cheaper than 3-way: about three times cheaper for CS_L1_C and four times cheaper for CS_L2_A.

CBC covers less control flow than 3-way for academic case studies and covers equally to 3-way for industrial case studies with one exception (Figure 9-14). Remarkably, ECBC has shown results similar to 3-way for all but one case study, i.e., CS_L1_C. For CS_L1_C the code coverage of the ECBC test suite is lower than the 3-way test suite for the reason explained in section 9.3.5. For all other case studies ECBC has control flow coverage equal to 3-way.

We conclude that the quality of a test suite created using CBC is equal to or lower than that of a test suite created using 3-way, i.e., lower mutation score and data flow coverage, lower or equal control flow coverage with a reduced cost. The quality of a test suite created using ECBC is equivalent to 3-way, i.e., similar mutation score, similar data and control flow coverage but at a much lower cost than a 3-way test suite.

9.5 Conclusion

The Category Partition technique is a specification based software testing technique [4]. Various selection criteria have been defined for this technique for generating test frames. Unlike other criteria, the Base Choice criterion is based on domain knowledge, on the expertise of the user and generates tests which are different from other criteria [13]. The importance of the Base Choice criterion and the limited literature on constraint handling with this criterion led to the current work.

The original definition of the Base Choice criterion (BC) does not account for complex constraints on choices. We extended the definition of BC in two different ways, referred to as Constrained Base Choice (CBC) and Extended Constrained Base Choice (ECBC), to combine choices strategically as in the original definition of BC while considering constraints. By construction, CBC subsumes BC and ECBC subsumes CBC.

We selected six case studies, three academic and three industrial, from the case studies described in Chapter 5, for the evaluation of the proposed criteria. The case studies varied in terms of size of the CP specification and number and complexity of constraints. We conducted 15 experiments and obtained 15 test suites for four out of six case studies. These test suites were obtained by n-way selection criteria, which used different technologies for test frame generation (CASA and ACTS), Base Choice and its variants (BC, CBC or ECBC) and used two different versions of CP specifications that differed in the presence and absence of single annotations. In order to draw assertive conclusions, we compared the quality of the test suites obtained from the proposed criteria (CBC, ECBC) with that of the existing criteria (Base Choice, Each Choice, Pairwise, 3-way combinations). We conducted primarily a quantitative analysis of cost

(i.e., number of test frames) and effectiveness at covering the source code (control and data flow) and at detecting faults, and complemented it with a qualitative analysis. We summarize our observations as follows:

- CBC and ECBC perform better than BC when a BC adequate test suite, because of inadequately handled constraints, is not Each Choice adequate. In case BC covers each choice of the CP specification, CBC performs similarly to BC, and ECBC continues to perform better. CBC and ECBC always perform better than Each Choice in terms of control and data flow coverage and fault detection, although at an additional, reasonable cost.
- For those case studies where it is easier to choose one appropriate choice as base choice in each category of the CP specification, ECBC performs better than Pairwise with better mutation score, control and data flow coverage with similar cost. ECBC performs equally to Pairwise where Each Choice is capable of 100% code coverage. For such CP specifications, ECBC performs equally to 3-way with similar mutation score, control and data flow coverage, though at a potentially significantly lower cost.
- We also observe that the quality of test frames obtained for a CP specification with Single annotations on select choices is better than when Single annotations are not used. We also observed a reduced cost when Single annotations are used in the CP specification for the Pairwise and 3-way criteria.
- We further performed a qualitative analysis and observed that some of the test frames generated by CBC and ECBC are never generated by Pairwise and on further investigation we discovered that they were generated by 3-way. Therefore,

we can conclude based on our study that CBC and more so ECBC exercise pairs and triplets one can find with 2-way and 3-way criteria, showing that the proposed criteria produce results similar to Variable Strength Covering Arrays [65] without incurring additional cost in terms of number of test frames.

- We used two combinatorial testing tools for the generation of test frames. We conclude that CASA, which is based on a metaheuristic technique, generates the same or smaller number of test frames than ACTS, which is based on a greedy algorithm.

Chapter 10 —Threats to validity

We foresee some threats to validity to the orchestrated survey that we performed in Chapter 4. We believe that the list of tools/algorithms we have identified (see complete list in appendices A.1 and A.2) is the most extensive one until March 2014, and definitely more extensive than the literature we surveyed at the time. We cannot however ignore the possibility of missing a tool or algorithm. One threat that we foresee in our work is that if a specific tool/algorithm is not compared, referred to or mentioned in a surveyed work, thesis or website there are chances that we have missed it. We however believe the risk is small since we captured publications by the main actors in the field.

While assigning a suitable category to a technique used by the algorithm for generating a combinatorial test suite, we encountered situations when the algorithms were not mentioned in detail or not mentioned at all. When we could not find the algorithms we have categorized them as “information not available”, and for some research papers, which lacked proper explanations, we made the nearest possible guess for the type of algorithm. Data may therefore not be entirely accurate. We however show there are a very few number of those occurrences and therefore the threat to our general observations and conclusions is small.

While looking for the maximum strength a tool supports we have considered two types of research work: work in which that tool/algorithm is proposed and the work in which that specific tool is used for comparison. Whichever strength is greater has been included in our analysis. We are aware of the fact that even while making an extensive search we might have missed some research work which would have given us a yet higher strength for a specific algorithm/tool. That can be a threat to the validity of our

work. In addition to that the tools/algorithms proposed after March 2014 have not been included. Last, we detailed our measurement framework and we believe our characterizations are robust enough to be reliable, thus leading to trustworthy results. Our report includes the search and analysis procedures, which should allow replications, for instance to update the search beyond 2014.

Similar to any experimental study our experimental work in Chapter 7 and Chapter 9, which is based on the academic and industrial case studies discussed in Chapter 5 and proposal of new criteria in Chapter 8, is also prone to validity threats. We discuss those threats in light of the framework proposed by Wohlin et. al. [129].

We foresee some internal validity threats to our work. This threat can occur when the outcome of the experiment is not impacted by the experimental set up but is caused by factors, which cannot be controlled during the experimental setup. The first internal validity threat we foresee is the non-deterministic results produced by CASA for test frame generation, i.e., each execution can lead to different test frames leading to better or worst results. All our results are based on one (the first) execution of CASA. We however used ACTS, which gives deterministic results, to compensate for CASA's non-determinism and we observed that results do not differ very significantly. We documented observed differences in results. Furthermore, we used different case studies, both academic in nature and provided by our industry partner, of varying complexity; we used different measures of test suite quality.

Another internal validity threat can be the selection of test inputs. We were very careful when selecting input values. We created priority classes to ensure the test inputs were selected as systematically as possible in an attempt to avoid introducing a bias. We

were very careful in reporting as many deviations as possible and explaining them through qualitative analyses. We however made some arbitrary decisions, although in permissible limits in Chapter 9, which resulted in varied mutation scores. We accompanied such results with explanations describing the reasons for those deviations. In fact, we were very careful in reporting as many deviations as possible. There could have been other factors, which we might have missed although we were very cautious when conducting the experiments.

There can be threats to conclusion validity in the present work. These threats correspond to the inability to draw valid conclusions. One reason for such a threat can be insufficient data to draw valid conclusions. In order to minimize this threat, along with academic case studies, which are well known to the testing community, we also selected industrial case studies provided by our industry partner Ericsson Inc. so that the results are outcome of a real scenario. These case studies have varied complexity in terms of constraints and the characteristics of CP specifications that reduces the threat to this validity. Further, in order to draw assertive conclusions, we compared our proposed criteria with the best quality test suite as discussed in Chapter 9, so our results are based on the outcome of four test suites rather than one and we concluded on the basis of trends observed from all of them. We also confirmed quantitative results with qualitative analysis.

Another threat to conclusion validity can be the variations in results that can be obtained when using CBC. Recall that during the prescriptive use of the criterion a decision is made at one point to make an initially infeasible test frame feasible by minimally changing it and there might be alternative series of changes to achieve this

goal. The alternatives may lead to different structural (control or data flow) coverage results, to different fault detection results; yet, using CBC only asks that one alternative be chosen. Only additional experiments will tell us the extent, if any, of this threat. There is no such threat when using ECBC since all the alternatives are investigated and considered. Another threat to conclusion validity can be the non-determinism of CASA. Different sets of test frames generated by CASA could have led to different conclusions. In order to combat this threat we used ACTS along with CASA which is representative of different technologies for the generation of test frames and concluded the results on the basis of more than one selection criteria (Each Choice etc.), used different surrogate measures for test suite quality and drove conclusions on the basis of similar trend.

Another threat to conclusions validity can be based on the implementation of the functionality and the outcomes of the test execution on the implemented functionality. If the source code is not robust to handle out-of-bound values either because of the requirement of the module or because of a poor design and implementation, the results and hence the conclusion can vary.

Construct validity is concerned with the way we defined our measurements. The number of test frames (i.e., test cases) and code coverage are reliable and established measurements of test suite quality. For measuring fault detection effectiveness, we used mutations. There can be concerns whether these mutants are representative of real faults. In order to reduce this threat we used both class and method level mutants. There is support in literature [118] which limits the likelihood of this threat [116]. Another threat can be the selection of equivalent mutants. We selected those mutants that were not killed by any test case as equivalent mutants. We also accompanied the decisions for the

selection for equivalent mutants with proper documentation where possible. The execution of our experiments using different selection criteria and different tools has led to a large number of test suites. Since all these test suites left alive the same set of mutants we argue that there are very weak chances that we have made an incorrect selection for equivalent mutants. This procedure to identify and discard equivalent mutants has been used by others before us [118] Another threat related to mutation analysis can be the presence of redundant mutants. Redundant mutants can result in an increased runtime during mutation analysis or can lead to a mutation score which is not representative of the actual number of mutants killed [130]. Our work compares the test suite obtained using different selection criteria. A redundant mutant, if present, will impact the result of every test suite but the comparison will still come out to be the same. Another possible threat to construct validity is related to the limited amount of structural coverage we were able to collect with the industry case studies. Fortunately, results from academic case studies, for which we have much more coverage information, overall match results from industry case studies. Another threat to the measurement of structural coverage is the use of different code coverage tool for academic and industrial case studies. Due to proprietary concerns, we could not use CodeCover for industrial case studies.

There can be threats to external validity as well. These threats are related to the external aspects of the experiments that can limit the generalizations of the results. This threat is mainly related to the case studies selected for performing the experiments and the CP specifications derived for them. For this reason, we selected academic and industrial case studies and found that the results are in agreement. Therefore, there are

good chances of generalization, though additional experiments are needed. Further, there can be various ways of deriving a CP specification for a system, for instance depending on the expertise of the tester. A tester can annotate different error or single choices based on a CP specification and the requirements of the system. We tried to be as comprehensive as possible while creating CP specifications and while assigning constraints among choices and selecting error and single choices. But there can still be differences in the decisions made by testers, which could lead to different results and hence different conclusions. We also acknowledge that the CP specifications can be improved to account for better code coverage and hence better results for one of the case studies i.e., NextDate. This improvement will be mainly to support robustness testing; which can be an outcome of the extension of the Category Partition technique.

We foresee one threat to external validity in the selection of base choices. In the definition of CBC and ECBC, we mentioned that the first test frame, which is formed with the combination of base choices, should be a valid test frame. However, in larger systems with many categories and choices, the tester/expert might select base choices that cannot be combined with one another. Although this goes against the intuition of the expert, we consider this as a threat to validity. In practice, a software tool can automatically detect an infeasible combination of base choices.

Chapter 11 –Conclusions and Contributions

The Category Partition (CP) method is a specification based software testing technique [4]. The process of applying CP begins by identifying parameters (and environment variables), categories and choices for a system under test. Since the technique is based on the expertise of the test engineer, one might begin by identifying a lot of categories and choices which when combined to form test frames generate an innumerable number of test frames which are infeasible if the choices are not competently constrained. The analysis of the quality of the test suite containing infeasible frames, in the present work, shows that they result in low fault detection, low code coverage and large cost. In order to combat this issue Ostrand and Balcer [4] suggest to introduce constraints among choices together with annotating some choices as Single or Error, since this will ensure that the test frames are not redundantly testing the same combinations which are perhaps infeasible.

But a question remains unanswered in the literature: "How much of these constraints (Error, Single and Constraints among choices) are enough?" We decided to answer this question.

Contribution 1: The basis of our evaluation is three industrial and two academic case studies, two selection criteria, two/three technologies for the generation of test frames and twelve experiments. We conclude that for defensive programming code if one constraint is to be used, Error annotations are most effective and if a tester decides on a pair of constraints, a pair of Single and Error annotations is most effective. For code with no defensive programming, Single annotations perform well regardless of the presence of constraints among choices, provided Single annotations are selected wisely. We also

conclude that the number of test cases contributing to test suite effectiveness, i.e., the number of frames contributing to the killing of mutants, increases as more constraints are introduced hence reducing the number of useless test frames.

Therefore, we conclude that, for software with or without good error handling code, specifying all kinds of constraints, i.e., Single, Error (for error handling code) and constraints among choices, certainly results in better quality (better fault detection, control and data flow coverage and reduced test suite cost) than without using these constraints at all. However, if a test engineer has limited resources and the functionality of the system allows her to meticulously annotate Single and Error choices, she is likely to obtain an equally effective test suite with these annotations.

Contribution 2: We also contribute by recognizing, based on one of our case studies, that the current definition of Category Partition technique is inadequate for extended robustness testing. We suggest that the CP technique can be extended by introducing specifications (categories and choices) for robustness, similar to the current definition where we have specifications based on the functionality. This extension can be similar to the extension of the use case model with negative use cases called Misuse cases [131].

Contribution 3: We conclude from the experiments that the cost of the test suite depends on three factors: **combination strength, number of choices per category and complexity of the constraints**. We conclude that specifying complex constraints and a large number of Single or Error choices results in an increased cost when the Each Choice (strength = 1) selection criterion is used. However, if the Pairwise criterion (strength = 2) is used with Single and Error annotations, the cost increases if there are

few choices per category and vice versa. If the Pairwise criterion is used with constraints among choices, the cost depends on two factors: the complexity of constraints in the CP specification and the number of choices per category. If there is more than one choice involved in four or more constraints and there is more than one n-tuple constraint, where n=>3, then there are chances that the number of test frames generated in the presence of constraints will be more than those generated in their absence. Further, if the number of choices per category is small, cost is high. This study can help a tester decide the tradeoff between the complexity of constraints and the size of the test suite.

Numerous selection criteria can be used for the generation of test frames as mentioned in literature [7, 47, 58]. Unlike other criteria, the Base Choice criterion is based on domain knowledge, on the expertise of the user and generates tests which are different from other criteria [13]. The importance of the Base Choice criterion and the limited literature on constraint handling with this criterion drives the next contributions.

Contribution 4: The Base Choice criterion uses the base choice for each category to form one test frame, which we refer to as base test frame. Other test frames are then created by moving away from the base choice, one choice at a time. This original definition of the criterion does not account for complex constraints. We extended the definition of the Base Choice (BC) criterion in two different ways, referred to as **Constrained Base Choice (CBC)** and **Extended Constrained Base Choice (ECBC)**, to combine choices strategically as in the original definition of BC while taking constraints into consideration. Briefly, the choices affected by constraints are also changed to their respective non-base choice in order to satisfy the constraints. There can be scenarios where the choices have complex constraints: e.g., ORs among the choices. CBC selects

any one candidate among the ORs such that it results in minimum change to other base choices. ECBC however selects all the OR candidates and forms multiple test frames. By construction, CBC subsumes BC and ECBC subsumes CBC. Also, CBC subsumes Each Choice.

Contribution 5: We selected six case studies, three academic and three industrial, for the evaluation of the proposed CBC and ECBC criteria. The case studies varied in terms of size of the CP specification and number and complexity of constraints. We conducted 15 experiments and obtained 15 test suites for each case study. These test suites were either obtained by n-way selection criteria, which used different technologies for test frame generation (CASA and ACTS) or Base Choice and its variants, i.e., BC, CBC or ECBC. We compared the results obtained from CBC and ECBC with the best results from two combinatorial testing tools (CASA and ACTS), when they were performing as combinatorial tools (in the absence of Single annotations) or when they are used to handle CP specific features (in the presence of Single annotations). We conclude that CBC and ECBC perform better than BC when a BC adequate test suite is not Each Choice adequate because of inadequately handled constraints. Further, we conclude that CBC and ECBC always perform better than Each Choice in terms of code (control and data flow) coverage and fault detection, although at an additional, reasonable cost.

For those case studies where it is easier to choose one appropriate choice as base choice in each category of the CP specification, ECBC performs better than Pairwise and equally to 3-way with similar mutation score and control and data flow coverage, though at a potentially significantly lower cost. However, ECBC performs equally to Pairwise when Each Choice is capable of 100% code (statement) coverage. From our qualitative

analysis we conclude that CBC and ECBC generate test frames which are present in Pairwise and 3-way adequate test suites hence showing results similar to variable strength covering arrays [65] without incurring additional cost in terms of number of test frames.

Contribution 6: Implementing the CP technique on large systems results in large number of categories (parameters) and choices (values). The choices are then combined, automatically, using Covering Array generation algorithms to generate test frames. Covering arrays come in various forms and have various capabilities and it is difficult to identify which covering array generation technique is the most suitable to the problem of generating test cases for the Category Partition technique. There are some characteristics specific to the CP technique and we could not find suitable literature to make informed decisions. Therefore, we performed an orchestrated survey on the available tools and algorithms for combinatorial test frame generation. Using our selection procedure, we obtained 75 tools and algorithms and categorized them on the basis of comparison criteria which are more specific to the CP technique. Our comparison framework allowed us to make a number of observations, e.g., on selection criteria support or constraint support of each tool or algorithm.

We observe that although metaheuristic, adaptive random/adhoc and algebraic techniques form a smaller part of the tool supporting covering array construction, they are equally focused on advance features for creating covering arrays as greedy based technologies. On the other hand, tools/algorithms based on greedy techniques are plenty in number, which can be attributed to the fact that they are flexible to implement. They support large system configurations including constraints, selection criteria, mixed

covering arrays and higher strengths, which is an essential requirement for software testing.

Contribution 7: The thesis also contributes a large number of experiments conducted using different case studies (academic and industrial), using different constraints representations (for different combinatorial testing tools), using different selection criterion and different test frame generation algorithms. We conclude, based on these experiments that, a metaheuristic technique generates a smaller number of test frames than a technique based on a greedy algorithm but in a greater amount of time, which was evident in the case study with complex constraints. We also conclude, similar to many before us, that the results obtained from ACTS are deterministic whereas those obtained from CASA are not and these results have an impact on the quality of the test suite.

Contribution 8: We also conclude on the implementation of the Category Partition technique on the industrial case studies that we obtained from Ericsson Inc. We implemented the CP technique on 11 real industrial case studies and compared the results of the CP generated test suite with the test suite created by the engineers at Ericsson using no formal specification based technique. We observed that the statement coverage improved with the implementation of the CP technique from 92.4 % to 95.6 %. On the other hand, the cost reduced drastically. The total number of test cases in the original test suite was 966 and with the CP technique the number of test cases came down to 81, which is a 91.6 % reduction in cost. We also conclude that since the industrial case studies have choices, which mostly take discrete values, the quality of the test suite can largely depend on the selection of a test frame generation algorithm.

Future Work

There is of course a lot that remains to be done to better understand the criteria we have introduced in this thesis as well as the Category Partition testing technique as a whole. Almost every aspect of its use, as illustrated earlier in the thesis in Figure 1-2, which we repeat in Figure 11-1 for convenience, should be subject to further investigations. Future work should at least include replications of the experiments we conducted for this thesis, attempting to address the threats to validity (e.g., additional case studies, consistent coverage measurement). Tools to support the (semi) automated use of CP should also be studied (decision 4 in Figure 11-1).

We have scratched the surface of a couple of variation points in the use of CP, specifically, criteria to produce test frames (decisions 3 in Figure 11-1). We have introduced two new criteria (CBC and ECBC) for the generation of test frames. The test suite obtained using these criteria show different n-way combinations. A future work can be a study of Pairwise and 3-way adequacy of the test suite obtained using the new criteria. Another extension to CBC and ECBC can be their support for Single annotations, so as a future work, CBC and ECBC can be extended to handle Single annotations. We have also suggested an extension of CP technique to support extended robustness testing. A future work can be such an extension for safety critical systems and more experiments to observe the results of the extension.

Another future work can be a study of the impact of the initial CP specification (decisions 1 and 2 in Figure 11-1) on the quality of the test suite. We predicted three factors that impact the cost of a test suite when Error annotations, Single annotations and constraints among choices are used in a CP specification. One of these factors is the

number of choices per category for which we did not give any accurate threshold value. Researching on accurate threshold values can be an extension to this work. The selection of input values satisfying the conditions imposed by test frames (decision 5 in Figure 11-1) can also be a future work, among other.

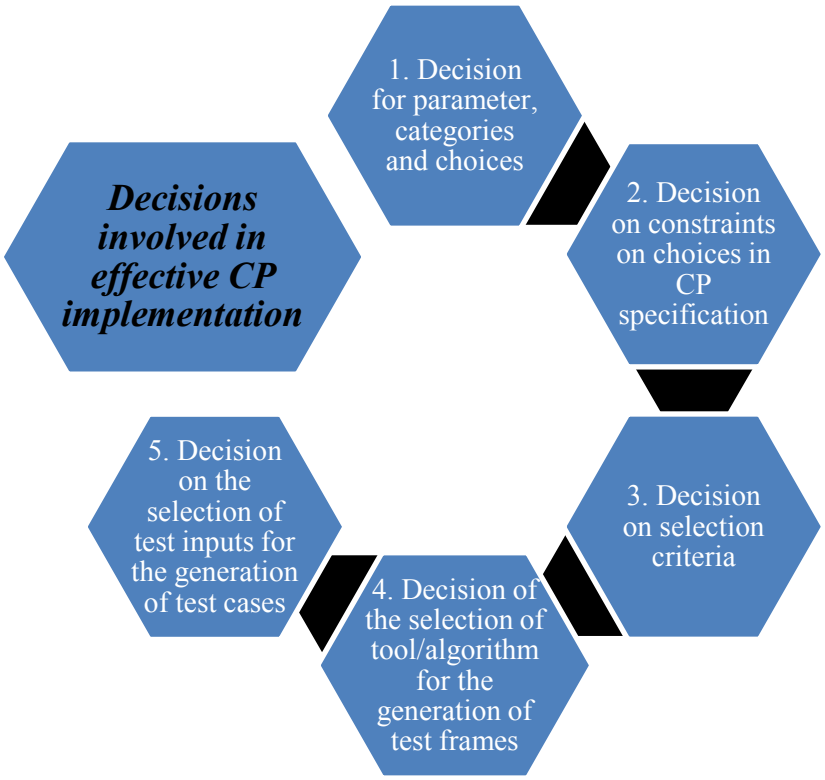


Figure 11-1 Different decisions made while implementing the Category Partition technique

Bibliography

1. Ammann, P. and J. Offutt, *Introduction to Software Testing*. 2008: Cambridge University Press.
2. Grochtmann, M. and K. Grimm, *Classification trees for Partition Testing.* Journal of Software Testing, Verification and Reliability, 1993. **3**(2): p. 63-82.
3. Malaiya, Y.K., *Antirandom Testing: Getting the most out of black box testing*, in *International Symposium on Software Reliability Engineering*. 1995: Toulouse, France. p. 86-95.
4. Ostrand, T.J. and M.J. Balcer, *The category-partition method for specifying and generating fuctional tests*, in *Communications of the ACM*. 1988. p. 676-686.
5. Myers, G.J., *The Art of Software Testing*. 1979: John Wiley & Sons.
6. Ammann, P. and J. Offutt, *Using formal methods to derive test frames in category-partition testing*, in *IEEE 9th Annual Conference on Computer Assurance*. 1994. p. 69-80.
7. Grindal, M., J. Offutt, and S.F. Andler, *Combination testing strategies: A survey.* Software Testing, Verification, and Reliability, 2005. **15**: p. 167-199.
8. Zhu, H. and P. Hall, *Test Data Adequacy measurement.* Software Engineering Journal, 1993. **8**(1): p. 21-29.
9. Balcer, M., W. Hasling, and T. Ostrand, *Automatic generation of test scripts from formal test specifications*, in *ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*. 1989: Nee York, NY, USA. p. 210-218.
10. Briand, L.C., Y. Labiche, Z. Bawar, and N.T. Spido, *Using Machine Learning to refine Category Partition test specifications and test suites*, in *8th International Conference on Quality software*. 2009. p. 1551-1564.
11. Labiche, Y. and F.R. Sadeghi. *Experimenting with Category Partition's 1-way and 2-way test selection criteria*. in *IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013.Luxemburg. p. 301-310.
12. Cohen, D.M., 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, 1997. **23**(7): p. 437-444.
13. Grindal, M., B. Lindström, J. Offutt, and S.F. Andler, *An Evaluation of Combination Strategies for Test Case Selection.* Empirical Software Engineering, 2006. **11**(4): p. 583-611.
14. Chan, E.Y.K., W.K. Chan, P.-L. Poon, and Y.T. Yu, *An empirical evaluation of several test-a-few strategies for testing particular conditions.* Software: Practice and Experience, 2012. **42**(8): p. 967-994.
15. Cohen, M.B., M.B. Dwyer, and J. Shi, *Interaction Testing of Highly-Configurable Systems in the Presence of Constraints.* International symposium on Software testing and analysis, 2007: p. 129-139.
16. Khalsa, S.K. and Y. Labiche, *An orchestrated survey on available algorithms and tools for Combinatorial Testing*, in *25th International Symposium on Software Reliability Engineering* 2014. p. 323-334.
17. Kuliamin, V.V. and A.A. Petukhov, *A survey of methods for constructing Covering Arrays.* Programming and Computer Software, 2011. **37**(3): p. 121-146.

18. Kuliamin, V. and A. Petukhov. *Covering array Generation Method Survey*. in *4th International Symposium on Leveraging Applications of formal methods, verification and validation*. 2010.Greece. p. 382-396.
19. Yu, L., Y. Lei, R.N. Kacker, and D.R. Kuhn. *ACTS: A Combinatorial Test Generation Tool*. in *6th IEEE International Conference on Software Testing, Verification and Validation*. 2013. p. 370-375.
20. Garvin, B.J., M.B. Cohen, and M.B. Dwyer. *An Improved Meta-heuristic Search for Constrained Interaction Testing*. in *1st International Symposium on Search Based Software Engineering*. 2009: IEEE Computer Society.Windsor. p. 13-22.
21. Czerwonka, J. *Pairwise testing in real world*. in *Pacific Northwest Software Quality Conference*. 2006. p. 419-430.
22. Hamming, W.R., *Error detecting and error correcting codes.* Bell Systems Technical Journal, 1950. **29**(2): p. 147-160.
23. Chen, T.Y., S.-F. Tang, P.-L. Poon, and T. Tse. *Identification of Categories and Choices in Activity Diagrams.* in *Fifth International Conference on Quality Software* 2005.Melbourne, Australia. p. 55-63.
24. Chen, T.Y., P.-L. Poon, and T.H. Tse, *A Choice Relation Framework for Supporting Category-Partition Test Case Generation.* IEEE Transactions on Software Engineering, 2003. **29**(7): p. 577-593.
25. Poon, P.L., T.Y. Chen, and T.H. Tse. *Choices, Choices: Comparing between CHOC'LATE and the Classification-Tree Methodology*. in *Ada- Europe Lecture notes in Computer Science*. 2012. p. 162-176.
26. Elmendorf, W.R. *Cause Effect Graphs in Functional Testing*. in *Share XLIII*. 1974.New York. p. 567-577.
27. Grindal, M. and J. Offutt. *Input Parameter Modeling for combination strategies*. in *25th conference on IASTED International Multi-conference*. 2007.CA, USA. p. 255-260.
28. Hartmann, J., M. Vieira, H. Foster, and A. Ruder. *TDE/UML: A UML-based Test Generator to Support System Testing*. in *International Software Testing Conference*. 2005.India.
29. Briand, L.C., M.D. Penta, and Y. Labiche, *Assessing and improving state based class testing: A series of Experiments.* IEEE Transactions on Software engineering, 2004. **30**(11): p. 770-793.
30. Hartmann, J., M. Vieira, H. foster, and A. Ruder, *A UML-Based approach to Software Testing.* Innovations in System and Software Engineering, 2005. **1**(1): p. 12-24.
31. Poon, P.L., T.Y. Chen, and T.H. Tse. *Incremental Identification of Categories and Choices for Test Case Generation: A Study of the Software Practitioners' Preferences*. in *IEEE. QSIC*. 2013.
32. Chen, T.Y., P.L. Poon, S.F. Tang, and T.H. Tse, *DESSERT: a DividE-and-conquer methodology for identifying categorieS, choiceS, and choicE Relations for Test case generation.* IEEE TSE, 2012. **38**(4): p. 794-809.
33. Chen, T.Y., P.L. Poon, S.F. Tang, and T.H. Tse, *On the Identification of Categories and Choices for Specification-Based Test Case Generation.* IST, 2004. **46**(13): p. 887-898.

34. Poon P.-L., T.T.H., Tang S.-F. and Kuo F. C., *Contributions of tester experience and a checklist guideline to the identification of categories and choices for software testing.* SQJ, 2011. **19**(1): p. 141-163.
35. Grindal, M., J. offutt, and J. Mellin. *Conflict management when using combination strategies for software testing*. in *ASWEC 2007*. 2007.Australia. p. 255-264.
36. Chan, E.Y.K., P.K. Poon, and Y.T. Yu. *On testing of particular input conditions*. in *28th annual International Computer Software and applications conference*. 2004.Hong Kong. p. 318-325.
37. Calvagna, A. and A. Gargantini, *A Logic-based approach to combinatorial testing with constraints.* Lecture Notes in Computer Science, 2008. **4966**: p. 66-83.
38. Calvagna, A. and A. Gargantini, *A Formal Logic Approach to Constrained Combinatorial Testing.* Journal of Automated Reasoning, 2010. **45**(4): p. 331-358.
39. Petke, J. *Constraints: the Future of combinatorial interaction testing*. in *IEEE/ACM SBST 2015*. 2015.NJ, USA. p. 17-18.
40. Bryce, R.C. and C.J. Colbourn, *Prioritized interaction testing for pairwise coverage with seeding and constraints.* Information and Software Technology, 2006. **48**(10): p. 960-970.
41. Cohen, M.B., M.B. Dwyer, and J. Shi, *Construction Interaction Test Suites for Highly Configurable Systems in the Presence of Constraints: A greedy Approach.* IEEE Transactions on Software Engineering, 2008. **34**(5): p. 633-650.
42. Petke, J. *Constraints: the Future of Combinational Interaction Testing*. in *IEEE/ACM 8th International Workshop on Search Based Software Testing*. 2015.Florance, Italy. p. 17-18.
43. Danziger, P., E. Mendelsohn, L. Moura, and B. Stevens, *Covering arrays avoiding forbidden edges.* Journal of Theoretical Computer Science Combinatorial Optimizations and Applications, 2009. **410**(52): p. 5403-5414.
44. Petke, J., M.B. Cohen, M. Harman, and S. Yoo, *Practical Combinatorial interaction testing: Empirical findings on efficiency and early fault detection.* IEEE TSE, 2015. **41**(9): p. 901-924.
45. Cohen, D.M., S.R. Dalal, A. Kajla, and G.C. Patton. *The automatic efficient test generator (AETG) system.* in *Fifth International Symposium on Software Reliability Engineering* 1994: IEEE Computer Society Press.Monterey, CA. p. 303-309.
46. Burr, K. and W. Young, *Combinatorial test techniques: Table-based automation, test generation and code coverage*, in *International Conference on Software Testing, Analysis, and Review*. 1998: San Diego, CA. p. 503-513.
47. Nie, C. and H. Leung, *A survey of combinatorial testing.* ACM Computing Surveys, 2011. **43**(2): p. 1-29.
48. Williams, A. and R. Probert. *A practical strategy for testing pairwise coverage of network interfaces*. in *7th International Symposium on Software Reliability Engineering*. 1996.White Plains, New York, USA. p. 246-254.
49. Othman, R.R. and K.Z. Zamli, *T-Way Strategies and Its Applications for Combinatorial Testing* International Journal on New Computer Architectures and Their Applications, 2011. **1**(2): p. 459-473.

50. Schroeder, P.J. and B. Korel. *Black-Box Test Reduction Using Input-Output Analysis*. in *2000 ACM sigsoft international symposium on Software Testing and Analysis* 2000.Portland, USA. p. 173-177.
51. Raaphorst, S., *Variable strength Covering arrays*. 2013, University of Ottawa: Ottawa.
52. Kuhn, D.R., Y. Lei, and R.N. Kacker, *Introduction to Combinatorial testing*. 2013: CRC Press. 341.
53. Garvin, B.J., M.B. Cohen, and M.B. Dwyer, *Evaluating improvements to a meta-heuristic search for constrained interaction testing.* Empirical Software Engineering, 2011. **16**(1): p. 61-102.
54. Hnich, B., S.D. Prestwich, E. Selensky, and B.M. Smit, *Constraint Models for the Covering Test Problem.* Journal on Constraints, 2006. **11**(2-3): p. 199-219.
55. Grindal, M., J. Offutt, and J. Mellin, *Handling Constraints in the Input Space when Using Combination Strategies for Software Testing*, in *Technical Report HS-IKI-TR-06-001*. 2006, School of Humanities and Informatics, University of Sk¨ovde.
56. Ahmed, B.S. and K.Z. Zamli, *A Review of Covering arrays and their applications to Software Testing.* Journal of Computer Science, 2011. **7**(9): p. 1375-1385.
57. Rahman, A., A. Al-Sewari, and K.Z. Zamli. *An Orchestrated Survey on T-Way Test Case Generation Strategies Based on Optimization Algorithms*. in *8th International Conference on Robotic, Vision, Signal Processing and Power Applications*. 2014. p. 255-263.
58. Anand, S., E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, and P. Mcminn, *An orchestrated survey of methodologies for Automated Software Test Case Generation.* Journal of Systems and Software, 2013. **86**(8): p. 1978-2001.
59. Bjorner, N., L.d. Moura, and C.M. Wintersteiger. *Microsoft Z3 Theorem Prover*. [cited; Available from: http://research.microsoft.com/en-us/projects/z3m/.
60. Laboratory, S.I.s.C.S. *Yices SMT solver*. [cited; Available from: http://yices.csl.sri.com/.
61. Fraser, G. and A. Arcuri. *Evo Suite: automatic test suite generation for object oriented software*. in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011.New York. p. 416-419.
62. Wegener, J., A. Baresel, and H. Sthamer, *Evolutionary test environment for automatic structural testing.* Information and software technology, 2001. **43**(14): p. 841-854.
63. Yilmaz, C., S. Fouché, M.B. Cohen, A. Porter, G. Demiroz, and U. Koc, *Moving Forward with Combinatorial Interaction Testing*, in *IEEE Computer Magazine*. 2014. p. 37-45.
64. Colbourn, C.J., *Combinatorial aspects of covering arrays.* Le Matematiche (Catania), 2004. **58**: p. 121-167.
65. Alsewari, A.R.A. and K.Z. Zamli, *Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support.* Information and Software Technology, 2012. **54**(6): p. 553-568.

66. Cohen, M.B., P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, and J.S. Collofello. *Variable Strength Interaction Testing of Components*. in *27th Annual International Computer Software and Applications Conference* 2003. p. 413-418.
67. Martinez, C., L. Moura, D. Panario, and B. Stevens, *Locating errors using ELAs, Covering arrays and adaptive testing algorithms.* Journal on Discrete Mathematics, 2009. **23**(4): p. 1776-1799.
68. Yilmaz, C., *Test Case-Aware Combinatorial Interaction Testing.* IEEE Transactions on Software Engineering, 2013. **39**(5): p. 684-706.
69. Demiroz, G. and C. Yilmaz. *Cost aware combinatorial interaction testing*. in *International Conference on Advances in System Testing and Validation lifecycle*. 2012. p. 9-16.
70. Kuhn, D.R., M.H. James, F.L. James, N.K. Raghu, and L. Yu. *Combinatorial Methods for Event Sequence Testing*. in *IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012: IEEE Computer Society.Montreal, QC, Canada. p. 601-609.
71. Sandro, F., B.C. Myra, and P. Adam. *Incremental covering array failure characterization in large configuration spaces*. in *18th International Symposium on Software testing and analysis*. 2009: ACM.USA. p. 177-188.
72. Turban, R.C., *Algorithms for covering arrays*. 2006, Arizona State University. p. 117.
73. Kitchenham, B. and S. Charters, *Guidelines for performing systematic literature reviews in software engineering.* , in *EBSE-2007-01.* 2007, Keele University.
74. Arksey, H. and L. O'Malley, *Scoping Studies: Towards a methodological framework.* International Journal of Social Research Methodology, 2005. **8**(2): p. 19-32.
75. Cohen, M.B., *Designing Test Suites for Software Interaction Testing*, in *Computer Science*. 2004, New Zealand: Auckland.
76. Al-Khiro, M.I.Y., *MIPOG: A Parallel T-Way Minimization strategy for combinatorial testing*, in *School of Electrical and Electronic Engineering*. 2010. p. 141.
77. Yu, L., *Advanced Combinatorial testing algorithms and applications*. 2013, The University of Texas at Arlington. p. 87.
78. Kuhn, D.R., R.N. Kacker, and Y. Lei, *Practical Combinatorial Testing*. 2010, National Institute of Standards and Technology.
79. Bryce, R.C., C.J. Colbourn, and D.R. Kuhn, *Finding Interaction Faults using distance based strategies*, in *18th IEEE International Conference on Engineering of Computer Based Systems*. 2011: Las Vegas, NV. p. 4-13.
80. The AETG Web Service Sciences, A.C., 2004 Available from: http://aetgweb.argreenhouse.com/. Access Date: May, 2014
81. Lei, Y., 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, 2008. **18**(3): p. 125-148.
82. Forbes, M., J. Lawrence, Y. Lei, R.N. Kacker, and D.R. Kuhn, *Refining the In-Parameter-Order Strategy for Constructing Covering Arrays.* Journal of Research of the National Institute of Standards and Technology, 2008. **113**: p. 287-297.

83. Yu, L., Y. Lei, M. Nourozborazjany, R.N. Kacker, and D.R. Kuhn. *An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation*. in *IEEE Sixth International Conference on Software, Testing, Verification and Validation*. 2013. p. 242-251.
84. Tai, K.-C. and Y. Lei, *A test generation strategy for pairwise testing.* IEEE Transactions on Software Engineering, 2002. **28**(1): p. 109-111.
85. Ahmed, B.S. and K.Z. Zamli, *A variable strength interaction test suites generation strategy using Particle Swarm Optimization.* Journal of System and Software, 2011. **84**(12): p. 2171-2185.
86. Bryce, R.C. and C.J. Colbourn, *One-test-at-a-time heuristic search for interaction test suites*, in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. 2007, ACM: London, England. p. 1082-1089.
87. Lei, Y. and K.C. Tai. *In-parameter-order: A test generation strategy for pairwise testing*. in *3rd IEEE International High Assurance System Engineering Symposium*. 1998.Washington, DC. p. 254-261.
88. Williams, A.W. *Determination of Test Configurations for Pair-Wise Interaction Coverage*. in *13th International Conference on Testing Communicating Systems: Tools and Techniques*. 2000: Kluwer, B.V.The Netherlands.
89. Williams, A.W. and R.L. Probert. *A measure for component interaction test coverage.* in *ACSI/IEEE International Conference on Computer Systems and Applications*. 2001: IEEE Computer Society Press.Beirut, Lebanon.
90. Cohen, M.B., P.B. Gibbons, M. W.B, and C.J. Colburn. *Constructing test cases for interaction testing*. in *25th International Conference on Software Engineering*. 2003.Portland, Oregon. p. 38-48.
91. Segall, I., R. Tzoref-Brill, and E. Farchi. *Using Binary Decision Diagrams for Combinatorial Test Design*. in *2011 International Symposium on Software Testing and Analysis*. 2011.Toronto, Canada. p. 254-264.
92. Toczki, J., F. Kocsis, T. Gyimothy, G. Danyi, and G. Kokoi, *SYS/3- A software Development tool.* Lecture Notes in Computer Science, 1991. **477**: p. 193-207.
93. Huang, R., X. Xie, T.Y. Chen, and Y. Lu. *Adaptive Random Test Case Generation for Combinatorial Testing*. in *IEEE 36th International Conference on Computer Software and Applications*. 2012.Izmir. p. 52-61.
94. Khatun, S., K.F. Rabbi, C.Y. Yaakub, and M.F.J. Klaib. *A Random Search Based Effective Algorithm for Pairwise Test Data Generation*. in *IEEE International Conference on Electrical Control and Computer Engineering*. 2011.Pahang, Malaysia. p. 293-297.
95. Allpairs Test Case Generation Tool, Bach, J., 1.2.1 2002 Available from: http://www.satisfice.com/tools.shtml. Access Date: Feb, 2014
96. Cohen, M.B., C.J. Colbourn, and A.C.H. Ling. *Augmenting Simulated Annealing to Build Interaction Test Suites*. in *Proceedings of the 14th International Symposium on Software Reliability Engineering* 2003: IEEE Computer Society.Los Almitos, CA. p. 394.
97. M. Chateauneuf and D.L. Kreher, *On the state of strength-three covering arrays.* Journal of Combinatorial designs, 2002. **10**(4): p. 217-238.
98. Hartman, A., *Software and Hardware Testing Using Combinatorial Covering Suites.* Graph Theory, Combinatorics and Algorithms, 2002. **34**: p. 237-266.

99. Tcases- A model driven test case generator, Project, M., 1.1.0 2012 Available from: https://code.google.com/p/tcases/. Access Date: March, 2014
100. ATD, Atyoursideconsullting, Available from: http://www.atyoursideconsulting.com/products/atd/atd_funcfeatures_tcg.html. Access Date: Feb,2014
101. PictMaster, Ltd, I.S.a.S.C., 5.7.3 2013 Available from: http://en.sourceforge.jp/projects/pictmaster/.
102. Zamli, K.Z., M.F.J. Klaib, M.I. Younis, N.A.M. Isa, and R. Abdullah, *Design and implementation of a t-way test data generation strategy with automated execution tool support.* Journal of Information Sciences, 2011. **181**(9): p. 1741-1758.
103. Othman, R.R., K.Z. Zamli, and L.E. Nugroho, *General variable strength t-way strategy supporting flexible interactions.* Maejo International Journal of Science and Technology, 2012. **6**(3): p. 415-429.
104. Othman, R.R. and K.Z. Zamli, *ITTDG: Integrated T-way test data generation strategy for interaction testing* Scientific Research and Essays, 2011. **6**(17): p. 3638-3648.
105. Jenny, Jenkins, B., 2005 Available from: http://www.burtleburtle.net/bob/math/jenny.html. Access Date: April, 2014
106. Calvagna, A. and A. Gargantini, *T-wise combinatorial interaction test suites construction based on coverage inheritance.* Journal of Software Testing, Verification, and Reliability, 2012. **22**(7): p. 507-526.
107. Li, L., Y. Cui, and Y. Yang. *Combinatorial Test Cases with Constraints in Software Systems* in *16th IEEE International Conference on computer supported cooperative work in design*. 2012.Wuhan. p. 195-199.
108. Zhao, Y., Z. Zhang, J. Yan, and J. Zhang. *Cascade: A Test Generation Tool for Combinatorial Testing*. in *IEEE 6th International Conference on Software Testing, Verification and Validation*. 2013.Luxemburg. p. 267-270.
109. Jorgensen, P.C., *Software Testing : A Craftsman's Approach*. 1995: CRC Press.
110. Marick, B., *The Craft of Software Testing*. 1995: PTR Prentice Hall.
111. J., H.K., V.D. S., C.J. J., and R.L. K., *A Practical Tutorial on Modified Condition/Decision Coverage*. 2001, NASA.
112. Smith, B.H. and L. Williams, *On guiding the augmentation of an automated test suite via mutation analysis.* Empirical software engineering, 2009. **14**(3): p. 341-369.
113. Ma, Y.S., J. Offutt, and Y.R. Kwon. *MuJava: A mutation system for Java*. in *28th International conference on software Engineering*. 2006.New York, USA. p. 827-830.
114. Yu-Seung, M. and O. Jeff. *Description of Class Mutation Operators for Java*. 2014: https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf.
115. Yu-Seung, M. and O. Jeff, *Description of Method-level mutation operators for Java*. 2011, https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf.
116. Andrews, J.H., L.C. Briand, Y. Labiche, and A.S. Namin, *Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria.* IEEE TSE, 2006. **32**(8): p. 608-624.
117. Offutt, A.J. and J. Pan, *"Detecting Equivalent Mutants and the Feasible Path Problem.* Software testing, verification and reliability, 1997. **7**(3): p. 165-192.

118. Andrew, J.H., L.C. Briand, and Y. Labiche, *Is mutation an appropriate tool for testing experiment*, in *ICSE 2005*. 2005. p. 402-411.
119. R. A. DeMillo, R. J. Lipton, and F.G. Sayward, *"Hints on Test Data Selection: Help for the Practicing Programmer.* IEEE Computer society, 1978. **11**(4): p. 34-41.
120. Chaim, M.L. and R.P.A.D. Araujo, *An efficient bitwise algorithm for intra-procedural data-flow testing coverage.* Information Processing Letters, 2013. **113**(8): p. 293-300.
121. Araujo, R.P.A. and M.L. Chaim, *Data flow testing in the large*, in *International conference on the Software Testing, verification and validation*. 2014: Cleveland, Ohio. p. 81-90.
122. Bitwise Algorithm - powered Definition-Use Association coverage, Andrioli, R., 0.3.0 2016 Available from: https://github.com/saeg/ba-dua/blob/master/README.md. Access Date: October 2016
123. Binder, R.V., *Testing object oriented systems - Models,Patterns and tools*. 1999: Addison Wesley.
124. Graves, T.L., M.J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, *An Empirical Study of Regression Test Selection Techniques.* TOSEM, 2001. **10**(2): p. 184-208.
125. CYGWIN, Cygnus, S., Available from: https://cygwin.com/index.html. Access Date: 2015
126. Cohen, M.B., M.B. Dwyer, and J. Shi. *Exploiting Constraint Solving History to Construct Interaction Test Suites*. in *TAICPART Mutation*. 2007.Windsor UK. p. 121-130.
127. zChaff, Zhang, D.L., 2007.3.12 2004 Available from: https://www.princeton.edu/~chaff/zchaff.html.
128. Khalsa, S.K., Y. Labiche, and J. Nicoletta. *The power of Single and Error annotations in Category Partition Testing: An Experimental Evaluation*. in *20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016)*. 2016.Limerick, Ireland. p. 28:1-28:10.
129. Wohlin, C., P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. 2012: Springer-Verlag Berlin Heidelberg.
130. Just, R., G.M. Kapfhammer, and F. Schweiggert, *Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?*, in *IEEE 5th international conference on Software testing, verification and validation (ICST 2012)*. 2012. p. 720-725.
131. Sindre, G. and A.L. Opdahl, *Eliciting security requirements with misuse cases.* Requirements Engineering, 2005. **10**(1): p. 34-44.
132. Tung, Y.-W. and W.S. Aldiwan. *Automating Test Case Generation for the New Generation Mission Software System*. in *In Proceedings of IEEE Aerospace Conference*. 2000. p. 431-437.
133. PICT Master, Ltd, I.S.a.S.C., 5.7.3 Available from: http://en.sourceforge.jp/projects/pictmaster/. Access Date: March, 2014
134. IBM Intelligent test case handler, Hartman, A., 1.0 2005 Available from: http://www.alphaworks.ibm.com/tech/whitch. Access Date: Dec 2013

135. Schroeder, P.J., J. Arshem, A.E. Kim, and P. Bolaki. *Combining Behavior and Data Modeling in Automated Test Case Generation*. in *3rd International Conference on Quality Software*. 2003: IEEE. p. 247-254.
136. TVG, Arshem, Available from: http://sourceforge.net/projects/tvg. Access Date: April 2014
137. Schroeder, P.J., P. Faherty, and B. Korel. *Generating Expected Results for Automated Black-Box Testing*. in *17th IEEE International conference on Automated Software Engineering*. 2002.Edinburgh, UK. p. 139-148.
138. Wang, Z., B. Xu, and C. Nie. *Greedy Heuristic Algorithms to Generate Variable Strength Combinatorial Test Suite* in *The Eighth International Conference on Quality Software*. 2008. p. 155-160.
139. Wang, Z., C. Nie, and B. Xu. *Generating Combinatorial Test Suite for Interaction Relationship* in *4th International Workshop on Software Quality Assurance*. 2007.Dubrovnik, Croatia. p. 55-61.
140. Sherwood, G.B. *Effective Testing of Factor Combinations*. in *Third International Conference on Software Testing, Analysis and Review*. 1994.Washington, DC.
141. Colbourn, C., M. Cohen, and R.C. Turban. *A Deterministic Density Algorithm for Pairwise Interaction Coverage*. in *IASTED International Conference on Software Engineering* 2004.Innsbruck Austria. p. 245-252.
142. Bryce, R.C. and C.J. Colbourn, *A Density-Based Greedy Algorithm for Higher Strength Covering Arrays.* Journal of Software Testing, Verification and Reliability, 2009. **19**(1): p. 37-53.
143. Wang, Z. and H. He, *Generating Variable Strength Covering Array for Combinatorial Software Testing with Greedy Strategy* Journal of Software, 2013. **8**(12): p. 3173-3181.
144. Ong, H.Y. and K.Z. Zamli, *Development of interaction test suite generation strategy with input-output mapping supports* Scientific Research and Essays, 2011. **6**(16): p. 3418-3430.
145. Kuhn, D.R., R.N. Kacker, and Y. Lei, *Practical combinatorial testing*, in *NIST Special publication*. 2010.
146. McCaffrey, J., *Pairwise Testing with QICT*, in *MSDN Magazine*. 2009.
147. Zhao, Y., Z. Zhang, J. Yan, and J. Zhang. *Cascade: A Test Generation Tool for Combinatorial testing*. in *6th IEEE International Conference on Software Testing, Verification and Validation Workshops*. 2013.Luxemburg. p. 267-270.
148. Klaib, M.F.J., K.Z. Zamli, N.A.M. Isa, M.I. Younis, and R. Abdullah. *G2Way - A Backtracking Strategy for Pairwise Test Data Generation* in *15th Asia-Pacific Software Engineering Conference*. 2008. p. 463-470.
149. Rabbi, K.F., A.H. Beg, and T. Herawan, *MT2Way: A Novel Strategy for Pair-Wise Test Data Generation.* Computational intelligence and intelligent systems, 2012: p. 180-191.
150. Rabbi, K.F., S. Khatun, C.Y. Yaakub, and M.F.J. Klaib, *EPS2Way: An Efficient Pairwise Test Data Generation Strategy* International Journal on New Computer Architectures and Their Applications, 2011. **1**(4): p. 1099-1109.
151. Chen, X., Q. Gu, J. Qi, and D. Chen. *Applying Particle Swarm Optimization to Pairwise Testing*. in *34th IEEE Computer Software and Applications Conference*. 2010.Seoul. p. 107-116.

152. Yuan, J., C. Jiang, and Z. Jiang. *Improved Extremal Optimization for Constrained Pairwise Testing*. in *International Conference on Research Challenges in Computer Science*. 2009. p. 108-111.
153. McCaffrey, J.D. *Generation of Pairwise Test Sets Using a Genetic Algorithm*. in *33rd Annual IEEE International Computer Software and Applications Conference*. 2009: IEEE Computer Society.Redmond, WA, USA. p. 626-631.
154. Shiba, T., T. Tsuchiya, and T. Kikuno. *Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing*. in *28th Annual International Computer Software and Applications Conference*. 2004.Los Alamitos, CA. p. 72-77.
155. Stardom, J., *Metaheuristics and the Search for Covering and Packing Arrays*. 2001, Simon Fraser University.
156. Flores, P. and Y. Cheon. *PWiseGen: Generating Test Cases for Pairwise Testing Using Genetic Algorithms* in *IEEE Conference on Computer Science and Automation Engineering*. 2011.Shanghai, China. p. 747-752.
157. Chen, X., Q. Gu, A. Li, and D. Chen. *Variable Strength Interaction Testing with an Ant Colony System Approach*. in *16th Asia-Pacific Software Engineering Conference*. 2009. p. 160-167.
158. LI, J., D. Xing, and Y. Zhao, *Combinatorial Test Suite Generation of Variable Strength Based on Harmony Search* Journal of Network and Information Security, 2013. **4**(2): p. 177-188.
159. Gonzalez-Hernandez, L., N. Rangel-Valdez, and J. Torres-Jimenez. *Construction of Mixed Covering Arrays of Variable Strength Using a Tabu Search Approach*. in *4th International Conference on Combinatorial optimization and applications* 2010. p. 51-64.
160. Younis, M.I., K.Z. Zamli, and N.A.M. Isa. *IRPS – An Efficient Test Data Generation Strategy for Pairwise Testing*. in *12th international conference on Knowledge-Based Intelligent Information and Engineering Systems*. 2008. Zagreb, Croatia. p. 493-500.
161. Younis, M.I. and K.Z. Zamli, *MIPOG - An Efficient t-Way Minimization Strategy for Combinatorial Testing* International Journal of Computer Theory and Engineering, 2011. **3**(3): p. 388-397.
162. Visual Pairwise Test Array Generator, VpTag, Available from: http://vptag.sourceforge.net/. Access Date: May, 2014
163. Williams, A.W., *Software Component Interaction Testing: Coverage Measurement and Generation of Configurations*, in *School of Information Technology and Engineering*. 2002, University of Ottawa: Ottawa.
164. Yan, J. and J. Zhang. *Backtracking algorithms and search heuristics to generate test suites for combinatorial testing*. in *Proceedings of the 30th Annual International Computer Software and Applications Conference* 2006. p. 385-394.
165. Yan, J. and J. Zhang, *A backtracking search tool for constructing combinatorial test suites.* Journal of Systems and Software, 2008. **81**(10): p. 1681-1693.
166. Bracho-Rios, J., J. Torres-Jimenez, and E. Rodriguez-Tello, *A New Backtracking Algorithm for Constructing Binary Covering Arrays of Variable Strength*, in *Proceedings of the 8th Mexican International Conference on Artificial Intelligence*. 2009: Guanajuato, México. p. 397-407.

167. Hartman, A. and L.P. Raskin, *Problems and algorithms for covering arrays.* Journal of Discrete Mathematics, 2004. **284**(1-3): p. 149-156.
168. TestCover, Sherwood, G., 2006 Available from: http://testcover.com/pub/constex.php. Access Date: Feb, 2014
169. Kobayashi, N., T. suchiya, and T. Kikuno, *A new method for constructing pair-wise covering designs for software testing.* Information Processing Letters, 2002. **81**(2): p. 85-91.
170. Hunter, J. *Hexawise tool*. [cited; Available from: https://hexawise.com/.
171. Pro-Test, SigmaZone, Available from: http://www.sigmazone.com/protest.htm. Access Date: Feb, 2014
172. SmartTest, Lewis, W.E., Available from: http://www.smartwaretechnologies.com/. Access Date: Feb, 2014
173. BenderRBT, INC., B., 8.0 1991 Available from: http://www.benderrbt.com/bendersoftware.htm. Access Date: Feb 2014
174. Tcases, Project, C., 2012 Available from: https://code.google.com/p/tcases/. Access Date: 19th May 2014
175. Horowitz, E., S. Sahni, and S. Rajasekaran, *Fundamentals of Computer algorithms*. 2000: W.H. Freeman and Company 761.

Appendix A An Orchestrated survey on the available algorithms and tools for combinatorial testing

A.1 List of tools/algorithms for generating test suites using combinatorial testing categorized on the basis of techniques

Generation Strategy	Greedy Techniques	Meta heuristic Techniques	Adaptive Random and Adhoc Techniques	Hybrid Techniques	Algebraic Techniques
Test Based Generation	• AETG Web Service [12, 75, 80] • Test Case Generator [132] • mAETG_SAT [15, 41] • ATGT[37] • PICT [21] • PictMaster[133] • Exhaustive search-Intelligent Test case handler (WHITCH)[134] • Jenny [105] • Test Vector Generator (TVG) [135, 136] • GVS [103] • Union [50] • Greedy [137] • Density [138] • ReqOrder in [139] • CATS [140] • Deterministic density algorithm [141] • Density Based Greedy [142] • DA-RO[143] • DA-FO [143] • ITTDG [104] • AURA [144] • Sequence Covering Array	• Particle Swarm Based Algorithm (OTAT)[151] • Extremal optimization based algorithm [152] • CASA [15, 20, 53] • Genetic Algorithms–GAPTS [153] • Genetic Algorithm Based • [154, 155] • GA based -PWiseGen[156] • Ant Colony algorithms[154] • Ant Colony System(ACS) [157] • Harmony search strategy [65] • Particle Swarm Test Generator VS-PSTG [85] • HSTCG [158] • Tabu Search [159]	• IRPS [160] • R2Way [94] • ART-CT [93] • Distance Based Technique [79] • AllPairs [95]	• Greedy Algorithm with Hill Climbing [86] • Greedy Algorithm with Simulated annealing [86] • Greedy Algorithm with Great Flood [86] • Greedy Algorithm with Tabu Search [86]	

Generation Strategy	Greedy Techniques	Meta heuristic Techniques	Adaptive Random and Adhoc Techniques	Hybrid Techniques	Algebraic Techniques
	Generator [145] • QICT [146] • Cascade [147] • IBM Focus [91] • VarDens [51] • G2way [148] • GTWay [102] • MT2Way [149] • EPS2way [150]				
Parameter based generation	• PairTest [84]. • ParaOrder [138] • ACTS [19] • tTuples [106] • CTWC [107] • MIPOG [161] • VpTag[162] • TConfig (IPO based) [163] • EXACT [164, 165] • Branch and Bound [166]	Particle Swarm Based Algorithm(OPAT) [151]		• IPOD (IPOG and Algebraic Technique)[81] • Augmented Annealing-combines Simulated Annealing and Algebraic Technique[75, 96]	• Tconfig [88] • Combinatorial Test Services (CTS) [167] • Test Cover [168] • Algebraic method [169]

A.2 Tools/algorithms found with no technical information

S.no	Name of Tool Algorithm	
1	T-Gen -SYS/3 - a Software Development Tool	[92]
2	Hexawise	[170]
3	ProTest	[171]
4	SmartTest	SmartTest [172]
5	ATD	[100]
6	BenderRBT	BenderRBT [173]
7	Tcases	[174]

A.3 Selection criteria supported by each tool/algorithm

S. No	Algorithm/tool	Each Choice	Base Choice	Variable Strength	Uniform strength	Input output based	Distance based	Random Input	All Combinations
1	AETG Web Service [12, 75, 80]				Yes				
2	PairTest [84].				Yes				
3	mAETG_SAT [15, 41]				Yes				
4	ATGT[37]				Yes				
5	ACTS [19]		Yes	Yes	Yes				
6	tTuples [106]				Yes				
7	Particle Swarm Based Algorithm(OTAT) [151]				Yes				
8	Extremal optimization based algorithm [152]				Yes				
9	CASA [15, 20, 53]			Yes	Yes				
10	Particle Swarm Based Algorithm (OPAT) [151]				Yes				
11	CTWC [107]				Yes				
12	PICT [21]		Yes (weights)	Yes	Yes				
13	MT2Way [149]				Yes				
14	EPS2way [150]				Yes				
15	IRPS [160]				Yes				
16	G2way [148]				Yes				
17	GTWay[102]				Yes				
18	Intelligent Test case handler (WHITCH)[134]			Yes	Yes				
19	Jenny [105]				yes				
20	Test Vector Generator (TVG) [135] [136]			Yes	Yes	Yes			
21	Tconfig [88]				Yes				
22	TConfig (IPO based) [163]				Yes				
23	GVS [103]			Yes	Yes	Yes			
24	Union [50]					Yes			
25	Greedy [137]			Yes	Yes	Yes			
26	ReqOrder in [139]					Yes			
27	Density [138]			Yes	Yes	Yes			
28	ParaOrder [138]			Yes	Yes	Yes			
29	Genetic Algorithms [154, 155]				Yes				
30	Ant Colony algorithms[154]				Yes				

S. No	Algorithm/tool	Each Choice	Base Choice	Variable Strength	Uniform strength	Input output based	Distance based	Random Input	All Combinations
31	Genetic Algorithm - GAPTS [153]				Yes				
32	Ant Colony System(ACS) [157]			Yes	Yes				
33	Greedy Algorithm with Hill Climbing [86]				Yes				
34	Greedy Algorithm with Simulated annealing [86]				Yes				
35	Greedy Algorithm with Great Flood [86]				Yes				
36	Greedy Algorithm with Tabu Search [86]				Yes				
37	Combinatorial Test Services (CTS) [167]				Yes				
38	Augmented Annealing-combines Simulated Annealing and Algebraic Technique[96]				yes				
39	IPOD (IPOG and Algebraic Technique)[81].				Yes				
40	CATS [140]				Yes				
41	Test Cover [168]				Yes				
42	Algebraic method [169]				yes				
43	Deterministic density algorithm [141]				Yes				
44	Density Based Greedy [142]				Yes				
45	DA-RO[143]			Yes	Yes	Yes			
46	DA-FO [143]			Yes	Yes	Yes			
47	Test Case Generator [132],				Yes				
48	R2Way [94]				Yes				
49	ART-CT [93]				Yes		Yes		
50	MIPOG [161]				Yes				
51	ITTDG [104]			Yes	Yes	Yes			
52	AURA [144]			Yes	Yes	Yes			
53	Harmony search strategy [65]			Yes	Yes *				
54	Particle Swarm Test Generator VS-PSTG [85]			yes	Yes*				
55	HSTCG [158]			Yes	Yes*				
56	EXACT [164, 165]				Yes				
57	Branch and Bound [166]			Yes*	Yes				
58	Tabu Search [159]			Yes*	Yes				
59	Distance Based Technique [79]				Yes		Yes	Yes	
60	T-Gen SYS/3 - a Software Development Tool [92]	Yes							
61	Sequence Covering Array Generator [145]				Yes				

S. No	Algorithm/tool	Each Choice	Base Choice	Variable Strength	Uniform strength	Input output based	Distance based	Random Input	All Combinations
62	Hexawise [170]			Yes	Yes				
63	QICT [146]				Yes				
64	Cascade [147]			Yes	Yes				
65	AllPairs [95]				Yes				
66	ProTest[171]				Yes				
67	VpTag[162]				Yes				
68	PictMaster[133]		Yes (weights)		Yes				
69	SmartTest [172]				Yes				
70	ATD [100]				Yes				
71	BenderRBT [173]				Yes				
72	IBM Focus [91]		Yes (weights)	Yes	Yes				
73	PWiseGen[156]				Yes				
74	VarDens [51]			Yes*	Yes				
75	Tcases [174]	Yes		Yes	Yes			Yes	

* Information Not Available

A.4 Maximum coverage strength support

S.No	Algorithm/tool	Maximum Strength support (t)	Configurations
1	AETG Web Service [12, 75, 80]	2	MCA(N, t, 4^1,3^{39}, 2^{35})
2	PairTest [84]	2	MCA(N, t, 4^1, 3^{39}, 2^{35})
3	mAETG_SAT [15, 41]	3	CCA(N, t, 2^{158}, 3^8, 4^4, 5^1, 6^1, F)
4	ATGT[37]	2	MCA(N, t, 4^1,3^{39}, 2^{35})
5	ACTS [19]	6	MCA(N, t, 10^2, 4^1, 3^2, 2^7)
6	tTuples [106]	6	MCA(N, t, 4^5, 2^{13})
7	Particle Swarm Based Algorithm(OTAT) [151]	2	MCA(N, t, 4^1, 3^{39}, 2^{35})
8	Extremal optimization based algorithm [152]	2	CCA(N, t, 2^{158}, 3^8, 4^4, 5^1, 6^1, t)
9	CASA [15, 20, 53]	3	CCA(N, t, 3^1, 2^4,F)
10	Particle Swarm Based Algorithm (OPAT) [151]	2	MCA(N, t, 4^1, 3^{39}, 2^{35})
11	CTWC [107]	5*	Information Not Available
12	PICT [21]	6	VSCA(N,3,3^{15}{CA(6,3^9)})
13	MT2Way [149]	2	CA(N, t, 3^4)
14	EPS2way [150]	2	CA(N, t, 3^4)
15	IRPS [160]	2	MCA(N, t, 5^1, 3^8, 2^2)
16	G2way [148]	2	MCA(N, t, 5^1, 3^8, 2^2)
17	GTWay(OTAT iterative) [102]	<=12	MCA(N, t, 10^2, 4^1, 3^2, 2^7)
18	Intelligent Test case handler (WHITCH)[134]	6	VSCA(N, 2, 10^1, 9^1, 8^1, 7^1, 6^1, 5^1, 4^1, 3^1 , 2^1, {MCA(6, 7^1, 6^1, 5^1, 4^1, 3^1, 2^1)})
19	Jenny [105]	<=8)	MCA(N, t, 10^2, 4^1, 3^2, 2^7)
20	Test Vector Generator (TVG) [135] [136]	6	VSCA(N, 2, 10^1, 9^1, 8^1, 7^1, 6^1, 5^1, 4^1, 3^1 , 2^1, {MCA(6, 7^1, 6^1, 5^1, 4^1, 3^1, 2^1)})
21	Tconfig [88]	2	MCA(N, t, 4^1, 3^{39}, 2^{35})
22	TConfig (IPO based) [163]	<=4	MCA(N, t, 10^2, 4^1, 3^2, 2^7)
23	GVS [103]	<=12	MCA(N, t, 10^2, 4^1, 3^2, 2^7)
24	Greedy [137]	3	MCA(N, t, 10^1, 6^2,4^3,3^1)
25	Density [138]	3	MCA(N, t, 10^1, 6^2,4^3,3^1)
26	ParaOrder [138]	3	MCA(N, t, 10^1, 6^2,4^3,3^1)
27	Genetic Algorithms- GAPTS [153]	2	MCA(N, t, 3^4, 3^{13}, 2^{100}, 10^{20})
28	Ant Colony algorithms(ACA) [154]	3	MCA(N, t, 10^1, 6^2,4^3,3^1)
29	Genetic algorithm based algorithm [154, 155]	3	MCA(N, t, 10^1, 6^2,4^3,3^1)

S.No	Algorithm/tool	Maximum Strength support (t)	Configurations
30	Ant Colony System(ACS) [157]	3	VSCA(N,2,3^{20},10^2, {MCA(3,3^{20},10^2)})
31	Greedy Algorithm with Hill Climbing [86]	4	MCA(N, t, 2^{10},3^3,4^2,5^1)
32	Greedy Algorithm with Simulated annealing [86]	4	MCA(N, t, 2^{10},3^3,4^2,5^1)
33	Greedy Algorithm with Great Flood [86]	4	MCA(N, t, 2^{10},3^3,4^2,5^1)
34	Greedy Algorithm with Tabu Search [86]	4	MCA(N, t, 2^{10},3^3,4^2,5^1)
35	Combinatorial Test Services (CTS) [167]	4	CA(N, t, 10^8)
36	Augmented Annealing-combines Simulated Annealing and Algebraic Technique[96]	3	CA(N, t, 14^{14})
37	IPOD (IPOG and Algebraic Technique)[81].	6	CA(N, t, 4^{15})
38	CATS [140]	3	CA(N, t, 6^4)
39	Test Cover [168]	4*	Information Not Available
40	Algebraic method [169]	2	MCA(N, t, 4, 3^{39}, 2^{35})
41	Deterministic density algorithm [141]	2	MCA(N, t, 4^1, 3^{39}, 2^{35})
42	Density Based Greedy [142]	6	CA(N, t, 5^{10})
43	DA-RO[143]	3	MCA(N, t, 10^1, 6^2, 4^3, 3^1)
44	DA-FO [143]	3	MCA(N, t, 10^1, 6^2, 4^3, 3^1)
45	Test Case Generator [132]	2 , t-wise*	MCA(N, t, 5^1, 3^8, 2^2)
46	R2Way [94]	2	CA(N, t, 3^4)
47	ART-CT [93]	4	MCA(N, t, 2^5, 3^5)
48	MIPOG [161]	11	MCA(N, t, 5^7, 2^4)
49	ITTDG [104]	12	MCA(N, t, 10^2, 4^1, 3^2, 2^7)
50	AURA [144]	3	MCA(N, t, 10^1,6^1,4^3,3^1)
51	Harmony search strategy [65]	14	VSCA(N,3,3^{15},{CA(14,3^{14})})
52	Particle Swarm Test Generator VS-PSTG [85]	6	VSCA(N,2,3^{15},{CA(6,7^1,6^1,5^1,4^1,3^1,2^1)})
53	HSTCG [158]	7	VSCA(N,2,4^3, 5^3, 6^2, {CA(7, 4^3, 5^3, 6^2)})
54	EXACT [164, 165]	5	CA(N, t, 2^6)
55	Tabu Search [159]	6	MCA(N, t, 2^2,3^2,4^2,5^2)
56	Branch and Bound [166]	5	CA(N, t, 2^6)
57	Distance Based Technique [79]	5*	Information Not available
58	Sequence Covering Array Generator [145]	4	80 events
59	Hexawise [170]	6	Information obtained via chat with tool support
60	QICT [146]	2	MCA(N, t, 3^4, 3^{13}, 2^{100}, 10^{20})
61	Cascade [147]	2*	Information not available
62	AllPairs [95]	2, N-Wise*	MCA(N, t, 5^1, 3^8, 2^2)

S.No	Algorithm/tool	Maximum Strength support (t)	Configurations
63	ProTest [171]	2*	Information Not Available
64	VpTag[162]	2*	Information Not Available
65	PictMaster[133]	6*	Information Not Available
66	SmartTest [172]	2*	Information Not Available
67	ATD [100]	2 *N-wise*	Information Not Available
68	BenderRBT [173]	2*	Information Not Available
69	IBM Focus [91]	2	MCA(N, t, 4^1,3^{39},2^{35})
70	PWiseGen[156]	2	MCA(N, t, 4^1,3^{39},2^{35})
71	Tcases [174]	3*	Information Not Available
72	VarDens [51]	4	CA(N, t, 5^{10})

*Information Not Available

A.5 Constraint handling support

S.No	Algorithm/tool	Representation of the constraint (*Forbidden tuples, allowed tuples or full constraint (logical expression*)	Constraint handling Mechanism 1.Constraints handled before executing test generation algorithm 2. Replacing the invalid test cases 3. Constraints handled by implementing an algorithm 4. Constraints handled using SAT Solvers
1	AETG Web Service [12, 75, 80]	Forbidden Tuples using if else expressions	Constraints handled by implementing algorithm
2	mAETG_SAT [15, 41]	Forbidden tuples converted into Boolean Formula	zChaff or MiniSAT SAT solver integrated into AETG algorithm. Solvers compute the constraints and AETG generates the test suites.
3	ATGT[37]	Full constraint support using prepositional logic	The combinatorial testing is represented as propositional logic problem including constraints (forbidden tuples) and SAL Constraint Solver is used to handle constraints and generate the test suite
4	ACTS [19] (IPOG-C [83]	Full constraint support using Boolean, relational and arithmetic operators based expressions	CHOCO Constraint Solver is integrated with the algorithm and is frequently called to handle constraints
5	tTuples [106]	Forbidden Tuples as Logical constraints	Greedy algorithm modified to handle constraints
6	Extremal optimization based algorithm [152]	Not enough information available	MiniSat Solver integrated with the Extremal Optimization algorithm
7	CASA [15, 20, 53]	Boolean expression in CNF	zChaff SAT Solver is integrated with the Simulated annealing algorithm
8	CTWC [107]	Forbidden tuples	Constraint handled by implementing an algorithm
9	PICT [21]	Full constraint support using Logical Expressions are used to define constraints	Forbidden tuples are obtained from logical expressions and then algorithm is implemented for handling constraints
10	Intelligent Test case handler (WHITCH) [134]	Forbidden tuples [15]	Information Not available
11	Jenny [105]	Forbidden tuples expressed as string of numbers and characters	Constraint handled by implementing an algorithm
12	Test Vector Generator (TVG) [135, 136]	Full Constraint support with Logical Expressions using relational operators	Constraint handled by implementing an algorithm
13	Combinatorial Test Services (CTS) [167]	Forbidden Tuples	Constraints handled by implementing an algorithm.
14	CATS [140]	Allowed Tuples	Constraints handled before executing test generation algorithm

S.No	Algorithm/tool	Representation of the constraint (*Forbidden tuples, allowed tuples or full constraint (logical expression*)	Constraint handling Mechanism 1.Constraints handled before executing test generation algorithm 2. Replacing the invalid test cases 3. Constraints handled by implementing an algorithm 4. Constraints handled using SAT Solvers
15	Test Cover [168]	Allowed Tuples	No description but it can be assumed that constraints are handled before giving to algorithm
16	Test Case Generator [132]	Full Constraint support as Logical Expressions	Constraints handled by implementing an algorithm
17	Harmony search strategy [65]	Information not available *	Constraints handled by implementing an algorithm
18	HSTCG [158]	Full constraint support. Paper discusses that the approach supports complex constraints with no further discussion	Constraints handled by implementing an algorithm
19	Distance Based Technique [79]	Forbidden tuples	Constraints handled by implementing an algorithm
20	T-Gen SYS/3 - a Software Development Tool [92]	Information not available *	No information available *
21	Sequence Covering Array Generator [145]	Forbidden tuples (excluded sequences)	No information available *
22	Hexawise [170]	Forbidden tuples	No information available *
23	Cascade [147]	Full constraint support using Boolean, relational and arithmetic operators based expressions	A pseudo-Boolean optimization (PBO) solver called clasp is used to handle constraints and optimize coverage. Constraint solving and optimization is integrated
24	ProTest [171]	Information not available*	Information not available*
25	VpTag[162]	Full constraint support. Paper discusses that the approach supports complex constraints with no further discussion	Information not available*
26	PictMaster[133]	Full constraint support using Logical Expressions are used to define constraints	Forbidden tuples are obtained from logical expressions and then algorithm is implemented for handling constraints
27	SmartTest [172]	Full constraint support using Logical Expressions are used to define constraints	Information not available*
28	BenderRBT [173]	Full constraint support using Logical Expressions are used to define constraints	Information not available*

S.No	Algorithm/tool	Representation of the constraint (*Forbidden tuples, allowed tuples or full constraint (logical expression*)	Constraint handling Mechanism 1.Constraints handled before executing test generation algorithm 2. Replacing the invalid test cases 3. Constraints handled by implementing an algorithm 4. Constraints handled using SAT Solvers
29	IBM Focus [91]	Full constraint support using Boolean Expressions in Java Syntax	Constraints handled by implementing an algorithm
30	Tcases [174]	Full constraint support. Properties are assigned to values and conditions are defined which are finally converted to Boolean expressions	Information not available*
31	AllPairs [95]	Information Not Available *	Information not available *
32	Augmented Annealing [75, 96]	Forbidden tuples	Constraints are handled before giving input to algorithms (as disjoint rows in the form of seeds)

*Information Not Available

A.6 Mixed covering array support

S.No	Algorithm/tool	Mixed Covering Array Support
1	AETG Web Service [12, 75, 80]	Yes
2	PairTest [84]	Yes
3	mAETG_SAT [15, 41]	Yes
4	ATGT[37]	Yes
5	ACTS [19]	Yes
6	tTuples [106]	Yes
7	Particle Swarm Based Algorithm(OTAT) [151]	Yes
8	Extremal optimization based algorithm [152]	Yes
9	CASA [15, 20, 53]	Yes
10	Particle Swarm Based Algorithm (OPAT) [151]	Yes
11	CTWC [107]	Information Not Available *
12	PICT [21]	Yes
13	MT2Way [149]	Yes
14	EPS2way [150]	Yes
15	IRPS [160]	Yes
16	G2way [148]	Yes
17	GTWay(OTAT iterative) [102]	Yes
18	Intelligent Test case handler (WHITCH)[134]	Yes
19	Jenny [105]	Yes
20	Test Vector Generator (TVG) [135, 136]	Yes
21	Tconfig [88]	Yes
22	TConfig (IPO based) [163]	Yes
23	GVS [103]	Yes
24	Union [50]	Yes
25	Greedy [137]	Yes
26	ReqOrder in [139]	Yes
27	Density [138]	Yes
28	ParaOrder [138]	Yes
29	Genetic Algorithms- GAPTS [153]	Yes
30	Ant Colony algorithms(ACA) [154]	Yes

S.No	Algorithm/tool	Mixed Covering Array Support
31	Genetic algorithm based algorithm [154, 155]	Yes
32	Ant Colony System(ACS) [157]	Yes
33	Greedy Algorithm with Hill Climbing [86]	Yes
34	Greedy Algorithm with Simulated annealing [86]	Yes
35	Greedy Algorithm with Great Flood [86]	Yes
36	Greedy Algorithm with Tabu Search [86]	Yes
37	Combinatorial Test Services (CTS) [167]	Yes
38	Augmented Annealing-combines Simulated Annealing and Algebraic Technique[96]	No
39	IPOD (IPOG and Algebraic Technique)[81].	Yes
40	CATS [140]	No
41	Test Cover [168]	Information Not Available *
42	Algebraic method [169]	Yes
43	Deterministic density algorithm [141]	Yes
44	Density Based Greedy [142]	Yes
45	DA-RO[143]	Yes
46	DA-FO [143]	Yes
47	Test Case Generator [132]	Yes
48	R2Way [94]	Yes
49	ART-CT [93]	Yes
50	MIPOG [161]	Yes
51	ITTDG [104]	Yes
52	AURA [144]	Yes
53	Harmony search strategy [65]	Yes
54	Particle Swarm Test Generator VS-PSTG [85]	Yes
55	HSTCG [158]	Yes
56	EXACT [164, 165]	Yes
57	Tabu Search [159]	Yes
58	Branch and Bound [166]	No
59	Distance Based Technique [79]	Information Not available
60	T-Gen SYS/3 - a Software Development Tool [92]	Information Not available
61	Sequence Covering Array Generator [145]	Information Not Available
62	Hexawise [170]	Information Not Available
63	QICT [146]	Yes

S.No	Algorithm/tool	Mixed Covering Array Support
64	Cascade [147]	Information not available
65	AllPairs [95]	Yes
66	ProTest[171]	Information Not Available
67	VpTag[162]	Information Not Available
68	PictMaster[133]	Information Not Available
69	SmartTest [172]	Information Not Available
70	ATD [100]	Information Not Available
71	BenderRBT [173]	Information Not Available
72	IBM Focus [91]	Yes
73	PWiseGen[156]	Yes
74	Tcases [174]	Information Not Available
75	VarDens [51]	No

Appendix B Category partition specification for academic case studies

B.1 Category partition specification of the Next Date case study

Parameters	Categories	Choices	Properties	Constraints	CASA	ACTS
Year	Range	[ch1.1] Year<1582		*[error]*		
		[ch1.2] 1582<=Year<=2100		[Base Choice]		
		[ch1.3] Year>2100		*[error]*		
	Type	[ch2.1] isLeapYear	[isLeap]	[Base Choice]		
		[ch2.2] isCommonYear	[isCommon]			
Month	Length	[ch3.1] 30Days (4, 6, 9, 11)	[has30Days]			
		[ch3.2] 31Days(except December) (1, 3, 5, 7, 8, 10)				
		[ch3.3] February (2)	[isFeb]	[Base Choice]		
		[ch3.4] December (12)	[isDec]			
		[ch3.5] month<1		*[error]*		
		[ch3.6] month>12		*[error]*		
Day	Range	[ch4.1] 1<=Day<28		*[single]*		
		[ch4.2] Day<1		*[error]*		
		[ch4.3] Day>31		*[error]*		
		[ch4.4] Day==29	[is29]	[Base Choice] [If !(isCommon&&isFeb)]	!is29 || !(isCommon && isFeb)) ➔ !is29||!isCommon||!isFeb	Day_Range="29"=>!(Month_Length ="Feb" && Year_type="common_year")
		[ch4.5] Day==30	[is30]	[If !isFeb]	!is30 || ! isFeb) ➔ !is30||!isFeb	Day_Range="30"=>!(Month_Length ="Feb")
		[ch4.6] Day==31	[is31]	[If !isFeb && !has30Days]	!is31 || (!isFeb && !has30Days) ➔ !is31||!isFeb && !is31||!has30Days	Day_Range="31" =>!(Month_Length="Feb" || Month_Length="30_days")
		[ch4.7] Day==28				

B.2 Category partition specification of the Triangle case study

Parameters	Categories	Choices	Properties	Constraints	CASA	ACTS
SA	[Cat 1] Values for SA	[ch1.1] SA > 0		[Base Choice]		
		[ch1.2] SA <=0		[error]		
	[Cat 2] SA compared to SB	[ch2.1] SA = SB	P21			
		[ch2.2] SA != SB	P22	[Base Choice]		
	[Cat 3] SA compared to SB and SC	[ch3.1] SA < SB + SC		[Base Choice]		
		[ch3.2] SA >=SB + SC	P32	[if !P21 && !P81 && !P62 && !P92] [single]	!p32||!p21&&!p32||!p81&&!p32||!p62&&!p32||!p92	SA_Compared_SB_and_SC="greater_equal"=>(!(SA_compared_SB="Equal") && !(SC_compared_SA = "Equal") && !(SB_compared_SA_and_SC = "greater_equal") && !(SC_compared_SA_and_SB="greater_equal"))
SB	[Cat 4] Values for SB	[ch4.1] SB> 0		[Base Choice]		
		[ch4.2] SB<= 0		[error]		
	[Cat 5] SB compared to SC	[ch5.1] SB = SC	P51			
		[ch5.2] SB != SC	P52	[Base Choice]		
	[Cat 6] SB compared to SA and SC	[ch6.1] SB < SA + SC		[Base Choice]		
		[ch6.2] SB > =SA + SC	P62	[if !P51 && !P21 && !P92)] [single]	!p62||!p21&&!p62||!p51&&!p62||!p92	SB_compared_SA_and_SC="greater_equal"=>(!(SC_compared_SA_and_SB="greater_equal") && !(SB_compared_SC = "Equal") && !(SA_compared_SB="Equal"))
SC	[Cat 7] Values for SC	[ch7.1] SC > 0		[Base Choice]		
		[ch7.2] SC <= 0		[error]		
	[Cat 8] SC compared to SA	[ch8.1] SC == SA	P81	[if !(P22 && P51) && !(P21 && P52)	!p81||!p22||!p51&&!p81||!p21||!p52	SC_compared_SA="Equal"=>(!(SA_compared_SB= "Not_Equal" && SB_compared_SC = "Equal") && !(SA_compared_SB = "Equal" && SB_compared_SC="Not_Equal"))
		[ch8.2] SC != SA	P82	[if !(P21 && P51) [Base Choice]	!p82||!p21||!p51	SC_compared_SA="Not_Equal"=> (!(SA_compared_SB = "Equal" && SB_compared_SC="Equal"))
	[Cat 9] SC compared to SA	[ch9.1] SC < SA + SB		[Base Choice]		
		[ch9.2] SC >= SA + SB	P92	[if !P51 && !P81	!p92||!p51&&!	SC_compared_SA_and_SB="greater_equal"=>

	and SB			[single]	p92||!p81	(!(SC_compared_SA = "Equal") && !(SB_compared_SC="Equal"))

B.3 Category partition specification of the PachHexChar case study

Parameter	Category	Choices	properties	Constraints	CASA	ACTS
RLEN	Value of RLEN	[1.1] less than zero	p11	[Error]		
		[1.2]zero	p12	If p23	!p12||p23	value_of_RLEn="p12"=>(number_of_hex_in_rlen="p23")
		[1.3]less than the string length	p13	[Base Choice] If p32	!p13||p32	value_of_RLEn="p13"=>(string_length="p32")
		[1.4]equal to the string length	p14	If p32	!p14||p32	value_of_RLEn="p14"=>(string_length="p32")
		[1.5]greater than the string length	p15	[Error]		
	Number of hex character in rlen	[2.1]Even	p21			
		[2.2]Odd	p22	[Base Choice]		
		[2.3]zero	p23			
Input string	Length	[3.1]Zero	p31			
		[3.2]greater than zero	p32	[Base Choice]		
	Type	[4.1]All Hexadecimal	p41	!p31&& !p23 && (p51||p52||p53||p54||p55)	!p41||!p31&&!p41||!p23&&!p41||p51||p52||p53||p54||p55	string_type="p41"=>!(string_length="p31") && !(number_of_hex_in_rlen="p23") && (string_chars="p51"||string_chars="p52"||string_chars="p53"||string_chars="p54"||string_chars="p55")
		[4.2]All Non Hexadecimal	p42	!p31 && p23 && p57	!p42||!p31&&!p42||p23&&!p42||p57	string_type="p42"=>!(string_length="p31") && (number_of_hex_in_rlen="p23") && (string_chars="p57")
		[4.3]Mixed Hexa and non hexa in RLEN	p43	[Base Choice] !p31 && !p23 && p56	!p43||!p31&&!p43||!p23&&!p43||p56	string_type="p43"=>!(string_length="p31") && !(number_of_hex_in_rlen="p23") && (string_chars="p56")
		[4.4] Null String	p44	p31 && p58	!p44||p31&&!p44||p5	string_type="p44"=>(string_length="p31")

					8	&& (string_chars="p58")
		[4.5] RLEN hex and !RLEN mixed	p45	!p31 && !p23 && p56 &&p13	!p45||!p31&&!p45||!p23&&!p45||p56&&!p45||p13	string_type="p45"=>!(string_length="p31") && !(number_of_hex_in_rlen="p23") && (string_chars="p56")&&(value_of_RLEn="p13")
		[4.6] RLEN non-hex and !RLEN mixed	p46	!p31 &&p23 && p56 &&!p14	!p46||!p31&&!p46||p23&&!p46||p56&&!p46||!p14	string_type="p46"=>!(string_length="p31") && (number_of_hex_in_rlen="p23") && (string_chars="p56")&&!(value_of_RLEn="p14")
	String characters	[5.1]Hex numbers	p51			
		[5.2]lowercase hex alphabets	p52			
		[5.3]uppercase hex alphabets	p53			
		[5.4]Mixed case hex alphabets	p54			
		[5.5]Mixed case hex alphabets and hex numbers	p55			
		[5.6]mixed hex and non hex char	p56	[Base Choice]		
		[5.7]non-hexadecimal character	p57			
		[5.8] Null String	p58			
	Last char in rlen	[6.1]Hex character	p61	[Base Choice] p41||p43 ||p45	!p61||p41||p43||p45	last_char_rlen="p61"=>(string_type="p41"||string_type="p43"||string_type="p45")
		[6.2]non hex char	p62	p42||p43 || p46	!p62||p42||p43||p46	last_char_rlen="p62"=>(string_type="p42"||string_type="p43"||string_type="p46")
		[6.3]none	p63	p12	!p63||p12	last_char_rlen="p63"=>(value_of_RLEn="p12")
ODD_ DIGIT	values	[7.1]<-1	p71	error		
		[7.2]>15	p72	error		
		[7.3]-1	p73	p21||p22||p23	!p73||p21||p22||p23	odd_digit="p73" =>(number_of_hex_in_rlen="p21"||number_of_hex_in_rlen="p22"||number_of_hex_in_rlen

						="p23")
		[7.4]number (0..9)	p74			
		[7.5]upper case/lowercase (10-15)	p75	[Base Choice]		

Appendix C Category Partition Specification for Industrial Case Studies

This appendix contains the CP specifications of the industrial case studies. We have replaced the original name of the parameter, categories and choices due to a non-disclosure agreement. The constraints among the choices are specified using the property names of the choices. Further, the CP specifications contain the constraints among the choices which are represented in two format. The first statement corresponds to the constraints as perceived by the tester and the second statement is the format that is given as input to CASA.

C.1 Category partition specification for Case Study CS_L1_A

Parameter	Category	Choice	Properties	Constraints
P1	Value	[1.1] ch_11	p11	
		[1.2] ch_12	p12	if (p51) [if !p12||p51]
		[1.3] ch_13	p13	if (p51) [if !p13||p51]
P2	Value	[2.1] ch_21	p21	
		[2.2] ch_22	p22	
		[2.3] ch_23	p23	
		[2.4] ch_24	p24	
		[2.5] ch_25	p25	[single]
		[2.6] ch_26	p26	
P3	Value	[3.1] ch_31	p31	p21 || p23 [if !p31||p21||p23]
		[3.2] ch_32	p32	
P4	Value	[4.1] ch_41	p41	if (p52 && p32 && p11 && p23) [if !p41||p52&&!p41||p32&&!p41||p11&&!p41||p23]
		[4.2] ch_42	p42	
P5	Value	[5.1] ch_51	p51	if (p23) && !p31 [if !p51||p23&&!p51||!p31]
		[5.2] ch_52	p52	
		[5.3] ch_53	p53	if (p23)&& p31 [if !p53||p23&&!p53||p31]

C.2 Category partition specifications for Case Study CS_L1_B

Parameters	Categories	Choices	Properties	constraints
P1	Value	[1.1] ch_11	p11	
P2	Value	[2.1] ch_21	p21	
		[2.2] ch_22	p22	
		[2.3] ch_23	p23	
		[2.4] ch_24	p24	
		[2.5] ch_25	p25	
P3	Value	[3.1] ch_31	p31	
		[3.2] ch_32	p32	
		[3.3] ch_33	p33	
		[3.4] ch_34	p34	
P4	Value	[4.1] ch_41	p41	if p421
		[4.2] ch_42	p42	
		[4.3] ch_43	p43	
P5	Value	[5.1] ch_51	p51	
		[5.2] ch_52	p52	
P6	Value	[6.1] ch_61	p61	
		[6.2] ch_62	p62	
		[6.3] ch_63	p63	
		[6.4] ch_64	p64	
P7	Value	[7.1] ch_71	p71	
		[7.2] ch_72	p72	
		[7.3] ch_73	p73	
P8	Value	[8.1] ch_81	p81	
		[8.2] ch_82	p82	
P9	Value	[9.1] ch_91	p91	
		[9.2] ch_92	p92	
P10	Value	[10.1] ch_101	p101	
		[10.2] ch_102	p102	
P11	Value	[11.1] ch_111	p111	
		[11.2] ch_112	p112	
		[11.3] ch_113	p113	
P12	Value	[12.1] ch_121	p121	
		[12.2] ch_122	p122	
		[12.3] ch_123	p123	
P13	Values	[13.1] ch_131	p131	
		[13.2] ch_132	p132	
		[13.3] ch_133	p133	
		[13.4] ch_134	p134	
P14	Values	[14.1] ch_141	p141	
		[14.2] ch_142	p142	
		[14.3] ch_143	p143	
P15	Values	[15.1] ch_151	p151	
		[15.2] ch_152	p152	
		[15.3] ch_153	p153	
P16	Values	[16.1] ch_161	p161	
P17	Values	[17.1] ch_171	p171	
P18	Values	[18.1] ch_181	p181	
		[18.2] ch_182	p182	if p332
P19	Values	[19.1] ch_191	p191	

Parameters	Categories	Choices	Properties	constraints
		[19.2] ch_192	p192	
P20	Values	[20.1] ch_201	p201	
P21	Values	[21.1] ch_211	p211	
		[21.2] ch_212	p212	
		[21.3] ch_213	p213	
		[21.4] ch_214	p214	
P22	Values	[22.1] ch_221	p221	
		[22.2] ch_222	p222	
		[22.3] ch_223	p223	
P23	Values	[23.1] ch_231	p231	
		[23.2] ch_232	p232	
		[23.3] ch_233	p233	
		[23.4] ch_234	p234	
P24	Values	[24.1] ch_241	p241	
		[24.2] ch_242	p242	if p332
P25	Values	[25.1] ch_251	p251	
P26	Values	[26.1] ch_261	p261	
P27	Values	[27.1] ch_271	p271	
		[27.2] ch_272	p272	
		[27.3] ch_273	p273	
		[27.4] ch_274	p274	
P28	Values	[28.1] ch_281	p281	
		[28.2] ch_282	p282	
		[28.3] ch_283	p283	
P29	Values	[29.1] ch_291	p291	
P30	Values	[30.1] ch_301	p301	p321
		[30.2] ch_302	p302	p322 && (p23 || p25)
P31	Value	[31.1] ch_311	p311	
P32	Values	[32.1] ch_321	p321	
		[32.2] ch_322	p322	
P33	Values	[33.1] ch_331	p331	
		[33.2] ch_332	p332	
P34	Values	[34.1] ch_341	p341	
		[34.2] ch_342	p342	
		[34.3] ch_343	p343	
P35	Values	[35.1] ch_351	p351	if p341
		[35.2] ch_352	p352	
P36	Values	[36.1] ch_361	p361	if p34
		[36.2] ch_362	p362	
P37	Values	[37.1] ch_371	p371	
P38	Values	[38.1] ch_381	p381	
		[38.2] ch_382	p382	
P39	Values	[39.1] ch_391	p391	
P40	Values	[40.1] ch_401	p401	
P41	Values	[41.1] ch_411	p411	
P42	Values	[42.1] ch_421	p421	
		[42.2] ch_422	p422	
P43	Values	[43.1] ch_431	p431	
		[43.2] ch_432	p432	
		[43.3] ch_433	p433	
P44	Values	[44.1] ch_441	p441	
		[44.2] ch_442	p442	

Parameters	Categories	Choices	Properties	constraints
P45	Values	[45.1] ch_451	p451	
		[45.2] ch_452	p452	
		[45.3] ch_453	p453	
P46	Values	[46.1] ch_461	p461	
		[46.2] ch_462	p462	

C.3 Category partition specifications for Case Study CS_L1_C

Parameter	Category	Choices	Properties	Constraints implemented using ACTS and CASA
P1	values	[1.1] ch_11	p11	
		[1.2] ch_12	p12	
		[1.3] ch_13	p13	
		[1.4] ch_14	p14	
		[1.5] ch_15	p15	
		[1.6] ch_16	p16	
P2	Values	[2.1] ch_21	p21	
		[2.2] ch_22	p22	
		[2.3] ch_23	p23	
P3	Values	[3.1] ch_31	p31	
		[3.2] ch_32	p32	
P4	Values	[4.1] ch_41	p51	ACTS constraints ➔P4="p51"=> (P5 = "p61") && !(P1="p11") CASA ➔!p51||p61&&!p51||!p11
		[4.2] ch_42	p52	P4="p52"=> (((P5 = "p63") &&(P1="p11"))||((P5 = "p62") && !(P1="p11"))) [single] !p52||p63||!p11&&!p52||p11||p62
		[4.3] ch_43	p53	P4="p53"=> (((P5 = "p64") && (P1="p11"))||((P5 = "p63") && !(P1="p11"))||((P5 = "p62") && !(P2="p21"))) !p53||p62||p64||!p11&&!p53||p64||!p11||!p21&&!p53||p62 ||p63||p11&&!p53||p63||p11||!p21
		[4.4] ch_44	p54	P4="p54"=> ((P5 = "p62") || ((P5 = "p63") && (P1="p11"))||((P5 = "p61") && !(P1="p11"))) !p54||p62||p63||!p11&&!p54||p61||p62||p11
		[4.5] ch _45	p55	P4="p55"=> ((P5 = "p62") ||(P5 = "p63")|| ((P5 = "p64") && (P1="p11"))||((P5 = "p61") && !(P1="p11"))) !p55||p62||p63||p64||!p11&&!p55||p61||p62||p63||p11
P5	Values	[5.1] ch_51	p61	
		[5.2] ch_52	p62	
		[5.3] ch_53	p63	
		[5.4] ch_54	p64	
E1	range	[E.1] ch_E_1	p71	

(E1 is an environment variable.)

C.4 Category partition specification for Case Study CS_L1_D

Parameters	Catgeories	Choices	Properties
P1	Values	[1.1] ch1.1	p11
		[1.2] ch_12	p12
P2	Values	[2.1] ch_21	p21
		[2.2] ch_22	p22
		[2.3] ch_23	p23
		[2.4] ch_24	p24
		[2.5] ch_25	p25
		[2.6] ch_26	p26
P3	values	[3.1] ch_31	p31
		[3.2] ch_32	p32
		[3.3] ch_33	p33
		[3.4] ch_34	p34
P4		[4.1] ch_41	p41
		[4.2] ch_42	p42
P5	values	[5.1] ch_51	p51
		[5.2] ch_52	p52
		[5.3] ch_53	p53
		[5.4] ch_54	p54
P6	values	[6.1] ch_61	P61

C.5 Category partition specification for Case Study CS_L1_E

Parameter	Category	Choice	Properties
P1	value	[1.1] ch_11	p11
		[1.2] ch_12	p12
P2	value	[2.1] ch_21	p21
		[2.2] ch_22	p22
P3	Value	[3.1] ch_31	p31
		[3.2] ch_32	p32

C.6 Category partition specifications of Case Study CS_L1_F

Parameters	Categories	Choices	Properties
P1	Values	[1.1] ch_11	p11
		[1.2] ch_12	p12
		[1.3] ch_13	p13
P2	Values	[2.1] ch_21	p21
		[2.2] ch_22	p22
		[2.3] ch_23	p23

C.7 Category partition specification for Case Study CS_L1_G

Parameters	Categories	Choices	Properties
P1	Values	[1.1] ch_11	p11
		[1.2] ch_12	p12
		[1.3] ch_13	p13
P2	Values	[2.1] ch_21	p21

		[2.2] ch_22	p22
		[2.3] ch_23	p23
P3	Values	[3.1] ch_31	p31
P4	Values	[4.1] ch_41	p41
		[4.2] ch_42	p42
		[4.3] ch_43	p43
P5	Values	[5.1] ch_51	p51
		[5.2] ch_52	p52

C.8 Category partition specification for Case Study CS_L1_H

Parameters	Categories	Choices	Properties
P1	Values	[1.1] ch_11	p11
		[1.2] ch_12	p12
		[1.3] ch_13	p13
P2	Values	[2.1] ch_21	p21
		[2.2] ch_22	p22
		[2.3] ch_23	p23
		[2.4] ch_24	p24
		[2.5] ch_25	p25
		[2.6] ch_26	p26
P3	Values	[3.1] ch_31	p31
		[3.2] ch_32	p32
		[3.3] ch_33	p33
		[3.4] ch_34	p34
		[3.5] ch_35	p35
		[3.6] ch_36	p36
P4	Values	[4.1] ch_41	p41
		[4.2] ch_42	p42
P5	Values	[5.1] ch_51	p51
		[5.2] ch_52	p52
		[5.3] ch_53	p53
		[5.4] ch_54	p54
		[5.5] ch_55	p55

C.9 Category partition specification for Case Study CS_L1_I

Parameter	Category	Choices	Properties
P1	Values	[1.1] ch_11	p11
		[1.2] ch_12	p12
		[1.3] ch_13	p13
P2	Values	[2.1] ch_21	p21
		[2.2] ch_22	p22
		[2.3] ch_23	p23
		[2.4] ch_24	p24
		[2.5] ch_25	p25
		[2.6] ch_26	p26
P3	range	[3.1] ch_31	p31
		[3.2] ch_32	p32
P4		[4.1] ch_41	p41
		[4.2] ch_42	p42

C.10 Category partition specification for Case Study CS_L1_J

Parameters	Categories	Choices	Properties
P1	Value	[1.1] ch_11	p11
		[1.2] ch_12	p12
		[1.3] ch_13	p13

C.11 Category partition specification for case study CS_L2_A

Parameters	Categories	Choices	Properties	Constraints
P1	Cat 1	[1.2] ch_12	p12	if(p37 ||p38 ||p39)
		[1.3] ch_13	p13	if(p37 ||p38 ||p39)
		[1.4] ch_14	p14	
		[1.5] ch_15	p15	
	Cat 2	[2.1] ch_21	p21	if (p12 || P13) && p41 && (p37 ||p38 ||p39)
		[2.2] ch_22	p22	if (p12 ||p13||p14) &&(p42) &&(p37 ||p38 ||p39)
		[2.3] ch_23	p23	if(p15)||(p14&&p41)) [!p23||p15||p14&&!p23||p15||p41]
P2	Cat 3	[3.1] ch_31	p31	
		[3.2] ch_32	p32	
		[3.3] ch_33	p33	
		[3.4] ch_34	p34	
		[3.6] ch_36	p36	
		[3.7] ch_37	p37	
		[3.8] ch_38	p38	
		[3.9] ch_39	p39	
P3	Cat 4	[4.1] ch_41	p41	
		[4.2] ch_42	p42	
	Cat 5	[5.1] ch_51	p51	if (!p41)
		[5.3] ch_53	p53	if p41
	Cat 6	[6.1] ch_61	p61	if (!p41)

Parameters	Categories	Choices	Properties	Constraints
		[6.2] ch_62	p62	if (!p41)
		[6.3] ch_63	p63	if (!p41)
		[6.4] ch_64	p64	if p41
P4	Cat 7	[7.1] ch_71	p71	
		[7.2] ch_72	p72	if (p32)
P5	Cat 8	[8.2] ch_82	P82	
P6	Cat 9	[9.1] ch_91	p91	
		[9.2] ch_92	p92	
		[9.3] ch_93	p93	
		[9.4] ch_94	p94	
		[9.5] ch_95	p95	
		[9.6] ch_96	p96	
P7	Cat 10	[10.1] ch_101	p101	if (p31)
		[10.2] ch_102	p102	if (p32|| p33)
		[10.3] ch_103	p103	if (p32 ||p34||p36||p37||p39)
		[10.4] ch_104	p104	if (p32 ||p38)
P8	Cat 11	[11.1] ch_111	p111	
		[11.2]ch_112	p112	
P9	Cat 12	[12.1] ch_121	p121	if p112
		[12.2] ch_122	p122	if (p111)
P10	Cat 13	[13.1] ch_131	p131	if (p94 ||p95||p96)
		[13.2] ch_132	p132	
P11	Cat 14	[14.1] ch_141	p141	
		[14.2] ch_142	p142	
	Cat 15	[15.1] ch_151	p151	p31||p32||p33
		[15.2] ch_152	p152	if (p34||p36||p37||p38||p39) && (!p141)

Appendix D Mutation Operators

D.1 Class level mutation operators

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess **m**odifier **c**hange
Inheritance	IHD	**H**iding variable **d**eletion
	IHI	**H**iding variable **i**nsertion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition
	IOR	**O**verriding method **r**ename
	ISI	**S**uper keyword **i**nsertion
	ISD	**S**uper keyword **d**eletion
	IPC	Explicit call to a **p**arent's **c**onstructor deletion
Polymorphism	PNC	**N**ew mothod call with **c**hild class type
	PMD	**M**ember variable **d**eclaration with parent class type
	PPD	**P**arameter cariable **d**eclaration with child class type
	PCI	Type **c**ast operator **i**nsertion
	PCC	**C**ast type **c**hange
	PCD	Type **c**ast operator **d**eletion
	PRV	**R**eference assignment with other comparable **v**ariable
	OMR	**O**verloading **m**ethod contents **r**eplace
	OMD	**O**verloading **m**ethod **d**eletion
	OAC	**A**rguments of **o**verloading method call **c**hange
Java-Specific Features	JTI	**t**his keyword **i**nsersion
	JTD	**t**his keyword **d**eletion
	JSI	**s**tatic modifier **i**nsersion
	JSD	**s**tatic modifier **d**eletion
	JID	Member variable **i**nitialization **d**eletion
	JDC	Java-supported **d**efault constructor **c**reation
	EOA	**R**eference **a**ssignment and contet assignment replacement
	EOC	**R**eference **c**omparison and content comparison replacement
	EAM	**A**ccessor **m**ethod change
	EMM	**M**odifier **m**ethod change

D.2 Method level mutation operators

Operator	Description
AOR	**A**rithmetic **O**perator **R**eplacement
AOI	**A**rithmetic **O**perator **I**nsertion
AOD	**A**rithmetic **O**perator **D**eletion
ROR	**R**elational **O**perator **R**eplacement
COR	**C**onditional **O**perator **R**eplacement
COI	**C**onditional **O**perator **I**nsertion
COD	**C**onditional **O**perator **D**eletion
SOR	**S**hift **O**perator **R**eplacement
LOR	**L**ogical **O**perator **R**eplacement
LOI	**L**ogical **O**perator **I**nsertion
LOD	**L**ogical **O**perator **D**eletion
ASR	**A**ssignment **O**perator **R**eplacement

Appendix E Complexity analysis of ECBC

In this appendix, we will perform a brief algorithmic analysis of ECBC. We already mentioned in the section 8.2 that ECBC is an extension of CBC which differs in how ECBC handles disjunction. ECBC takes into consideration all the disjunct clauses whereas CBC ensures that either one of the clauses is satisfied. Since ECBC subsumes CBC, we will perform the analysis for ECBC.

The performance of an algorithm is quantified by the number of steps or operations that algorithm will require for its execution and the memory used by the algorithm as the inputs increase. This is called the time and space complexity of the algorithm [175]. This performance analysis is based on the asymptotic behaviour of the algorithm and does not require the computation of the exact run time or exact number of operations of the algorithm. An asymptotic behaviour can be a lower bound which represents the best case or an upper bound which represents the worst case scenario of an algorithm. We will use the Big-O asymptotic notation for the analysis of the ECBC algorithm.

The Big-O notation, written as $f(n) = O(g(n))$, gives the asymptotic upper bound of the algorithm being analyzed. The function $f(n) = O(g((n))$ ($f(n)$ is a Big-O of $g(n)$) if and only if there exist positive constants c and n_o such that $f(n) \leq c * g(n)$ for all n, where $n \geq n_o$. Function $g(n)$ provides an upper bound for the value of $f(n)$ which is an approximation of the worst case scenario i.e., the maximum number of operations required by an algorithm for its execution. The Big-O notation, which is the order of magnitude function, therefore, focuses on the most dominant part of $f(n)$. For example, consider an algorithm with the following $f(n) = n^3 + 3n^2 + 45$. For the large values of n the most dominant part will be n^3, therefore $n^3 + 3n^2 + 45 = O(n^3)$ as $n^3 + 3n^2 + 45 \leq 10n^3$ for $n \geq 2$.

Figure E.1 sketches the basic structure of the ECBC criterion. Similar to greedy algorithms, our algorithm also follow the test based strategy for the generation of test frames, i.e., it generates one test at a time while following the ECBC technique. In Figure E.1, column 3, we calculate the Big-O as we proceed.

Assume that there are N categories in the CP specification and each category has K choices. K_{mp} is the p_{th} choice of category m and K_{mb} is its base choice (Table E.1).

The process begins with the base test frame, line 1, figure E.1. As mentioned earlier, we assume that the base test frame is feasible and a good starting point for the ECBC criterion. The algorithm is repeated for all the categories (Line 2) and for K-1 non-base choices in each category (line 3), since one choice in each category is already involved in the base test frame. In lines 4 and 5, we change a base choice to a non-base choice in the base test frame and check the constraints involved with the non-base choice and the remaining base choices in the base test frame. At this point we are assuming that all the constraints, i.e., the constraint in the non-base choice and the constraints with other choices do not conflict with each other (line 6, figure E.1). A conflicting constraint represents a problem with the CP specification and the specification will need to be changed. For example, consider the base test frame ch_14, ch_23, ch_34, ch_41, ch_103, ch_152 from section 8.3.1. Assume that ch_152 has been changed to ch_151 and the constraint with ch_151 is p31 ||p32 ||p33 (Table 8-I). Assume that ch_31, ch_32 and ch_33 further has a constraint !ch_151. This would be an issue with the CP specification and the constraints would need to be reconsidered.

Line 7 will check the infeasibility of the test frame resulting from the change of the base choice of the m_{th} category to the non-base choice. For example, changing ch_152 to

Table E.1. Synthetic CP specification

Category	Choice	Constraint
N1	K11	K21&&K31…&&Km1
	K1b	
	…	
	K1p	
N2	K21	K11||K1b||...||K1p|| K31||K3b||...||K3p|| Km1||Kmb||...||Kmp
	K2b	
	…	
	K2p	
N3	K31	
	K3b	
	…	
	K3p	
…	…	…
Nm	Km1	
	Kmb	
	…	
	Kmp	

ch_151 in the base test frame ch_14, ch_23, ch_34, ch_41, ch_103, ch_152 will introduce infeasibility because ch_151 can only be combined with p31 ||p32 ||p32 and none of these choices are base choices (base choice is ch_34) as shown in Table 8-I. We can use a SAT solver to check this feasibility. We will however not include the complexity of calling a SAT solver in the complexity of this algorithm. If the test frame is infeasible this is necessarily due to the constraints associated with the new non-base choice.

As defined in Figure 8-2 there can be three types of constraints; conjunctions, disjunctions and conjunctions of disjunctions. Since we are considering the worst-case scenario, we will consider the maximum of each kind of constraints, which will be an assumption for calculating the Big-O of the ECBC algorithm.

Considering the constraint types separately (line 8); in the worst-case scenario, a conjunction in the constraint involved with the non-base choice can include the non-base

choices of all the categories. Therefore, to make the test frame feasible, the base choices of all the N-1 categories should be changed to non-base choices in order to satisfy the constraints. We change N-1 categories because the non-base choice of one of the categories is already the basis of this constraint which was changed in line 4. For example, assume a constraint on ch_22 as 12 && 31 && 42 && 101 && 151 (Table 8-I) or consider the constraint on K11 in the table E.1. Observe that none of the choices involved in this constraint is a base choice; therefore, we will have to change the base choice of all n-1 categories to non-base choices. Recall that we are assuming the constraints are feasible (line 6) so this change will not result in any conflict.

Consider the second constraint of type disjunction; which is handled in line 12. A constraint can have multiple choices joined together with the disjunction operator (||). In ECBC we will generate multiple test frames by satisfying all these options but one at a time, as compared to CBC where one test frame which satisfies any one of the options is sufficient. This constraint handling in the presence of disjunctions makes our ECBC algorithm different from other combinatorial test frame generation algorithms specifically CASA and ACTS, which attempt to satisfy any one of the disjunctive clause while generating a test frame.

Discussing the complexity of ECBC, which will be an upper bound and considering the worst-case scenario, we are assuming that a disjunctive constraint associated with a choice of category m will have choices from all the other categories joined together with a disjunction. The total number of choices in that disjunctive constraint would be $\sum_{i=1}^{N-1}(k_i)$, which is exactly the number of times loop from lines 15 to 20 will execute. Although this is not a realistic situation and we did not observe such a constraint in any

of our case studies, it is ideal for calculating the upper bound. Therefore we are assuming that all K choices from N-1 categories, in lines 13 and 14, will be involved in a disjunctive constraint and these disjunctions are between choices of same and different categories (refer steps 6.i(a), 6.i(b) Figure 8-2), the same is also shown as an example on choice K21 in table E.1.

Line 15 will be a single operation which will satisfy the first disjunctive clause and will change the related base choice to a non-base choice. This change can result in an infeasible test frame because of the constraint associated with the most recent non-base choice change and the remaining base choices. Lines 16 to 19 will therefore ensure that the feasibility is checked and in case the test frame is infeasible a change from a base choice to a non-base choice is accordingly done. There will be N-2 base choices, associated with N-2 categories, that will be involved in the feasibility check and, in case of infeasibility, there will be at most K-1 choice changes to make the test frame feasible. There are N-2 categories because choices of the two categories resulting from line 4 and 15 have already been changed, i.e., the first non-base choice and the change of choice related to the first disjunctive clause. We now need to check if the latter change introduced any conflicts with the remaining base choices. In case of infeasibility, the base choices can be changed to K-1 other non-base choices. So the operations will proceed (K-1)*(N-2) times in a worst case scenario.

The feasibility of the test frames can be checked with the help of a SAT solver. Literature [126] shows that SAT solvers like zChaff [127] can be used to check the feasible and such solvers work on the CNF format, similar to the format we are dealing

with currently. Similarly to the previous case, we do not include the complexity of the solvers in this study.

Figure 8-2 shows that the third constraint type can be the conjunctions of disjunctions. Practically we cannot foresee and generalize the extent to which the choices will be involved in conjunctions and disjunctions in this type of constraint in order to calculate the number of operations required for handling this case. To generalize the percentage of conjunction and disjunctions in CNF is a topic of future research. However, in order to resolve this constraint, the algorithm shall follow steps 13 to 20 when the constraints contains disjunctions and steps 8 to 11 when the constraint has conjunctions. So our calculation of Big-O notation for ECBC procedure is based on the worst case scenario related to the cases when the CP specification has constraints which are either all conjunctions or all disjunctions.

The time complexity of the algorithm is greatly influenced by the number of categories, number of choices in each category and the constraints among the choices. We have however kept the constraints constant and at their maximum value and have calculated the Big-O of ECBC algorithm on the basis of categories and choices. The third column in figure E.1 shows the Big-O calculation for each step as per the previous discussion. We have ignored the constants in the calculation process. Disjunctions between the choices have a greater impact on the time complexity of the algorithm as compared to conjunctions, since they increase the number of operations required to obtain an ECBC test suite.

The Big-O calculation in the case of a conjunction is $N*(K-1)*(N-1) => N^2K-N^2-NK+N$

Considering the most dominant term, the upper bound for conjunction would be $O(N^2K)$.

In case of disjunctions, ECBC will be more complex at line 18.

Therefore the Big-O calculation is = $N*(K-1)*\sum_{i=1}^{N-1}(k_i)*(N-2)*(K-1)$. --(1)

We simplify the number of operation in lines 13 and 14 i.e., $\sum_{i=1}^{N-1}(k_i)$ => (N-1)*K.

Therefore equation (1) becomes = N*(K-1)*(N-1)* K *(N-2)*(K-1)

$$=N^3K^3 + N^3K - N^32K^2 - 3N^2K^3 - 3N^2K + 6N^2K^2 + 2NK^3 + 2NK - 4NK^2$$

Considering the most dominant term, the upper bound for disjunction would be $O(N^3K^3)$.

Figure E.1. Complexity analysis of ECBC algorithm

	Algorithm	Big-O
1	Make the first test frame with all the base choices assuming that all the constraints are satisfied.	1
2	Repeat for all N categories {	N
3	Repeat for all K-1 non-base choices in each category {	N*K-1
4	Change base choice of m_{th} category to non-base choice in the base test frame	N*(K-1)
5	Check the constraints involved with the new non-base choice with the constraints of other choices	
6	While the constraints are feasible {	
7	While test frame is infeasible {	
8	If the constraint of the non-base choice contains conjunctions	
9	For all K_{N-1} choices in the constraint {	
10	Change the k_{mbth} choice in the test frame to non-base choices to satisfy the constraints. }	N*(K-1)*(N-1)
11	Generate test frame }	
12	If constraint of the non-base choice contains disjunctions	
13	Repeat for all N-1 categories involved in the constraints {	
14	Repeat for all K choices involved in the constraint {	N*K-1*
15	Change the related base choice in the test frame to the choice mentioned in disjunctive clause	$\sum_{i=1}^{N-1}(K_i)$ N*K-1
16	Repeat for remaining n-2 base choices in the test frame {	$\sum_{i=1}^{N-1}(K_i)$ *N-2
17	While test frame is infeasible {	
18	change the base choice of mth category to the remaining k-1 non-base choices. }	N*K-1 $\sum_{i=1}^{N-1}(K_i)$ *N-2* K-1
19	Add choice to the test frame }	
20	Generate test frame }	
	} } } }	