Comparison of Coverage Criteria for the Category Partition Method using Automatically Generated Test Suites with Melba

by

Faezeh Rafsanjani Sadeghi

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

Master of Applied Science

in

Department of Systems and Computer Engineering

Carleton University
Ottawa, Ontario

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-93509-5

Our file Notre référence
ISBN: 978-0-494-93509-5

Abstract

Software testing can be an expensive task especially when facing tight deadlines. Melba was introduced to ease the task of re-engineering a test suite by using the Category Partition black box testing technique and a machine learning algorithm. In addition to re-engineering a test suite, there is a need for automated generation of test suites using this tool and Category Partition. We have introduced a new feature to Melba to allow this task, while supporting three selection criteria specifically defined for the Category Partition method: Base Choice, Each Choice and Pair Wise criteria. More specifically, we automatically generate test suite specifications that satisfy these criteria, that is sets of test cases specifications. These specifications can then be used to create concrete test cases, with actual test data values. Since there has been little work done on the comparison of these criteria, we have conducted an experimentation using five case studies to compare the Base Choice, Each Choice and Pair-Wise criteria in terms of cost and effectiveness. Cost was measured in terms of number of test cases in adequate test suites while effectiveness was measured in terms of fault detection using automatically generated mutants.

Acknowledgments

I would like to take this opportunity to thank the people who have encouraged, supported and guided me through my work. I am mostly grateful to God for giving me the strength, wisdom and patience to finish my thesis.

I can't thank my dear supervisor, Professor Labiche, enough for his help in every step of completing this work. I will always be grateful for having a wise and knowledgeable supervisor who has supported me through this academic experience.

To my dear husband Hossein for his unconditional love and support. I am very grateful for having such an understanding person in my life and I would not have been able to complete this work without his encouragement and support.

I would like to thank my dear parents for raising me, helping me in every step of the way in life and teaching me that you can never stop learning. Lastly I would like to thank my dear brother Farzad who lights up my life with his encouragement and love.

Contents

List of Tables

List of Figures

LIST OF ACRONYMS

Acronym	Explanation
AETG	Automatic Efficient Test Case Generator
ATC	Abstract Test Case
ATS	Abstract Test Suite
CA	Covering Array
CASA	Covering Arrays by Simulated Annealing
CCA	Constrained Covering Aarrays
CIT	Combinatorial Interaction Testing
CP	Spec Category-partition Specification
DT	Decision Tree
IPO	In Parameter Order
ISP	Input Space Partitioning
MC/DC	Modified Condition/Decision Coverage
ML	Machine Learning
OEC	Output equivalence class
SUT	System Under Test
TC	Test Case
TS	Test Suite
TSL	Test Specification Language

1 INTRODUCTION

Software testing is an expensive component of software development and maintenance [1]. It requires up to 50% of software development cost and this cost is even more for safety-critical applications [2]. With this being said, it can sometimes be more expensive to not test software and face a critical issue. Therefore software testing is an essential part of the software development cycle.

In order to test software in an efficient way, different testing techniques and associated criteria are used to decide when to stop testing. These techniques include defining a criterion to be used. A few examples are Graph Coverage, Logic Coverage, Input Space Partitioning and Syntax-Based Testing.

One black box testing technique is Input Space Partitioning (ISP), or Category Partitioning. A Black box testing technique refers to deriving tests from external sources including specification and requirements. ISP has the advantage of being easy to use because the tester does not need to understand the implementation and everything is based on a description.

ISP divides the input space of the program under test into partitions where each partition represents a parameter and is based on equivalence class partitioning and boundary value analysis. The parameters are divided into Categories and Choices. Then constraints can be defined to specify that certain choices have to be used together or should never be used together. Two other specific types of constraints allow the user to specify that a choice should only be used once (the *single* constraint) or that a choice will necessarily result in an erroneous behaviour (the *error* constraint). We refer to the set of parameters, categories and choices and constraints to as the Category Partition Specifications (CP specification). A test frame is composed of a set of choices where one choice is chosen from each category and a set of test frames is known as a test suite. More precisely, 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 executed. Constraints can be defined to restrict the combination of choices that can make a test frame or to avoid unfeasible combinations.

A few criteria are used to select combinations of choices for test frames in test suites. Three criteria that we are going to look at include Base Choice, Each Choice and Pair Wise. With the Base Choice criterion a base choice is selected for each of the categories. A base choice is the most "important" choice for a category that is to be tested more often than other choices. A first test frame is created by using all the base choices, i.e. the base choice for each category. Other test frames are then chosen by holding all but one base choice constant and using each non-base choice once for the one non-constant choice. With the Each Choice criterion, each choice in every category must appear in at least one test frame. And lastly with the Pair Wise criterion, an adequate test suite exercises each possible pair of choices from different category at least once. [2]

Little is known about the cost effectiveness of the above mentioned criteria. To evaluate the cost and effectiveness of the criteria we have to resort to experiments since this cannot be done by analytical means.

Melba is a tool used for re-engineering test suites using ISP [18]. The user can define the CP specification using this tool and the tool automatically generates an abstract test suite for a test suite provided as input by the user. A machine-learning algorithm is then used to analyze the abstract test suite and suggest improvements to the user.

1.1 Problem Statement

The work on Melba has been mostly on re-engineering test suites using ISP. In this context Melba performs well but there is a need for other functionalities that would make Melba more applicable and helpful. One of these includes the building of test suites that are adequate for various criteria using automated means.

In addition to the lack of automation in terms of building test suites, little is known about the respective cost and effectiveness of the selection criteria used for building test suites with the Category Partition method. Adding automated test suite generation capability to Melba then would allow us to benefit from the Category Partition support Melba already offers and then perform experiments.

1.2 Contributions:

In this thesis we make contributions in two ways. The first is by adding tool support to Melba for building selection criteria for Category Partition, and the second contribution is to investigate the cost and effectiveness of these criteria. These contributions are summarized below.

1. Added tool support: Added tool support to Melba for automatic generation of test suites (i.e., test suite specifications) using a CP specification.
2. Use of Criteria: Used three selection criteria for test suite generation which are Base Choice, Each Choice and Pair Wise.
3. Experimentation: Performed five case studies where I evaluate the cost and effectiveness of each of the three selection criteria

1.3 Thesis Organization

The remaining of this Thesis is structured as follows: In section 2 we discuss the related work. Section 3 discusses the existing Melba method and our contribution in terms of tooling. In section 4 we discuss the design of case studies and selection of test cases. Section 5 is a discussion of the results obtained for each case study. We then conclude the thesis in section 6 with the discussion of the contributions and results in terms of method and case study experiments.

2 RELATED WORK

There have been some approaches that relate to Melba and the experimentation of use of some criteria in building test frames. However as we shall see none of these works directly compares to Melba.

In [3] the authors describe a language called Test Specification Language (TSL) which is used for writing formal test specifications, i.e., CP specifications. In this paper, the specification of software is written in TSL and the tool generates a test script from these specifications. There is only a mention of an automatic tool generator for generating test frames (implemented as scripts) using the TSL tool. There is no mention of selection criteria used to generate test frames, other than exercising all possible combinations, whereas we generate test frames using one of the Base Choice, Each Choice and Pair Wise selection criteria. The TSL language can be regarded as a text-based version of the graphical user interface provided by Melba.

In [4] the authors use the Base Choice criterion for deriving test frames from CP specifications written in TSL [3]. This paper makes a contribution by defining the Base Choice criterion and demonstrates the feasibility of applying the criterion to an example specification. It only gives a brief explanation of the Each Choice criterion and why it should not be used. In our thesis we compare three different criteria in terms of cost and effectiveness by means of experimentation.

In [5], based on a Category Partition specification (TSL format), the authors empirically compare the test-once approach and the test-a-few approach . The test-once approach is to exercise each choice in the CP specification once (and only once), which is similar to the Each choice criterion, whereas the test-a-few approach is to exercise choices several times (n times, n>1). The authors argues in favour of the latter approach because the uniformity assumption being made when creating the choice, i.e., the corresponding equivalent class, may not hold, and therefore trying the choice several times (with different values from the domain defined by the choice) is an attempt to deal with the risk of the uniformity assumption to not hold. Different strategies to involve a choice several times are discussed: With the *random sampling* strategy, n the test cases involving the choice can be randomly selected; With the *choice coverage* strategy, test cases are randomly selected such that the choice is combined with every other choice at least

once (if possible); With the choice-occurrence sampling strategy, test cases are randomly selected such that the choice is combined with every other choice at least a certain number of times (whenever possible). It is worth noting that, although the authors discuss test cases, at that stage, these are more likely test frames. As such, it is very possible to involve a specific choice in several test frames but select one (and only one) value from the domain specified by the choice in all the test cases derived from those test frames; This would not reduce the risk of the uniformity assumption to not hold. From an experiment involving one CP specification and many implementations of the specification created by students, the authors conclude that the test-a-few approach provides a good cost-benefit ratio: it requires a small number of test cases and improves fault detection. Although this paper makes a great contribution, there is no mention of tooling involved and the paper's focus is not on the three criteria discussed in our thesis.

In [6] the authors develop a choice relation framework for supporting category-partition test case generation. They construct a choice relation table using the CP specification, i.e., a n times n table where n in the number of choices, that specifies which pairs of choice are necessarily used together, cannot be used together, or may be used together in a test frame. Next, the authors construct a choice priority table, which ranks choices in order of priority of occurrence in test frames. Last, test frames are created from the two tables. The focus of the paper is to provide a systematic way to define constraints among choices, to provide mechanism for checking for consistency among constraints and to provide an effective test frame construction process. This paper does not focus on different selection criteria and there is no experimentation with fault detection.

[7] is an analysis and discussion of common mistakes testers make when they use an ad hoc approach to identify categories and choices from informal specifications. The authors conducted three empirical case studies to identify these mistakes and suggest ways to help the user to avoid these mistakes. Although this paper is related to category partitioning, the focus is on how to make better use of this method.

The classification tree method [8] is an alternative to the CP method for black-box testing. It is also based on boundary value analysis and equivalence class partitioning but essentially presents the specification of the problem in a tree-like structure, which is not the case of the CP method. The authors discuss two selection criteria, referred to as maximality and minimality

criteria, which are equivalent to the All-combinations and Each choice criteria we discussed earlier. Tool support for the classification tree method and those two criteria also exist [9, 10]. It supports the Each choice, Pair-wise, Three-wise (triplets of choice) and All combinations criteria [10]. Other differences exist between the classification tree method and the category partition method [11]. To the best of our knowledge, no empirical comparison of the selection criteria for the classification tree method exists.

To sum up, our thesis is different from the related work in one of the following ways: 1) Melba (now) provides tool support for the generation of test suites (i.e., sets of abstract test cases) whereas some of the related work does not; 2) Generation of test suites is done by some of these related works but none is generating test suites for the three criteria which are Base Choice, Each Choice and Pair Wise; 3) We have conducted experiment with five different systems to compare the three criteria mentioned above, whereas there is no experiment done in this context in the related works.

3 TOOL SUPPORT

In this section we will give a brief discussion on the purpose and functionality of Melba and what it currently supports: section 3.1. Then we will discuss what needs to be done in terms of tool support for the purpose of our thesis: section 3.2. The next step is to give a discussion on what has been added to Melba in order to meet our needs for the conduction of experiments: section 3.3.

3.1 Melba Current Functionality

Melba is currently used as a tool for analyzing test suites, refining test suites or category partition specifications to improve the test suite and/or the category specification. In order to understand what the functionality of Melba is we are going to give a brief explanation of the Melba process: Figure 1.

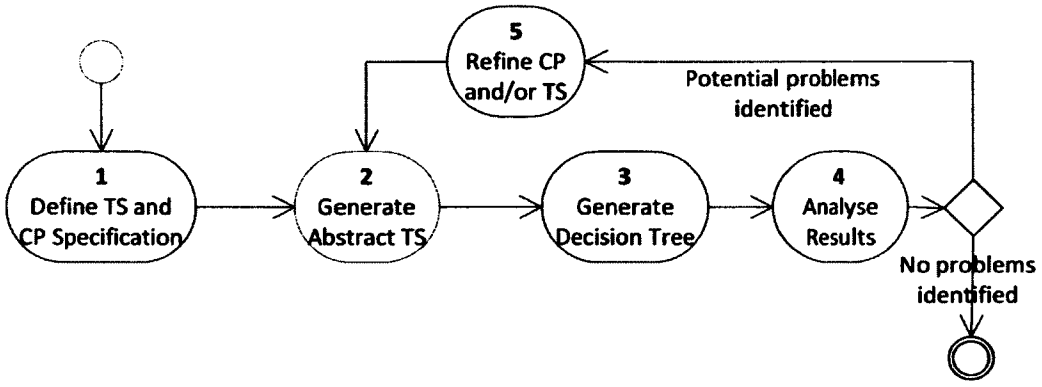

Figure 1 The Melba Process [12]

The user provides two pieces of information in the first step. This information includes a test suite (TS) that is used for evaluation and possible improvements, and the specification of the System Under Test (SUT). The specification is based on the Category-partition (CP) method [13] including inputs and environment variables and their categories and choices, as well as constraints on choices. Output equivalence classes (OEC) are also defined for further decision tree analysis.

In step 2 an abstract test suite (ATS) is generated which means a set of Abstract Test Cases (ATC) is created where each ATC corresponds to exactly one Test Case (TC) in the user's test

suite. Melba identifies which category and choice correspond to the values of all parameters for a given test case. In this step there is early identification of some problems in the CP Specification as illustrated in Figure 2.

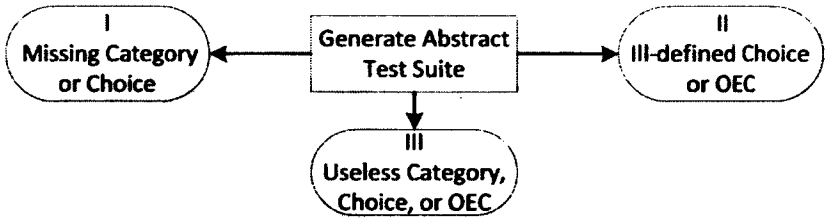

Figure 2 Potential problems identifiable when generating ATS [12]

The next step in the process is the generation of a decision tree (DT) based on the ATS built with the user's input information. A Machine Learning (ML) algorithm named C4.5 classification tree algorithm ([14], [15]) is used in the generation of the DT. The user can select an output parameter of interest so that the algorithm can identify input-output relationships between combination of pairs (category, choice) and output equivalence classes for that parameter.

After the generation of the DT in step 2, the user can analyze the DT results to identify, with some guidance provided by Melba, potential problems in the test suite or the CP specification (Figure 3).

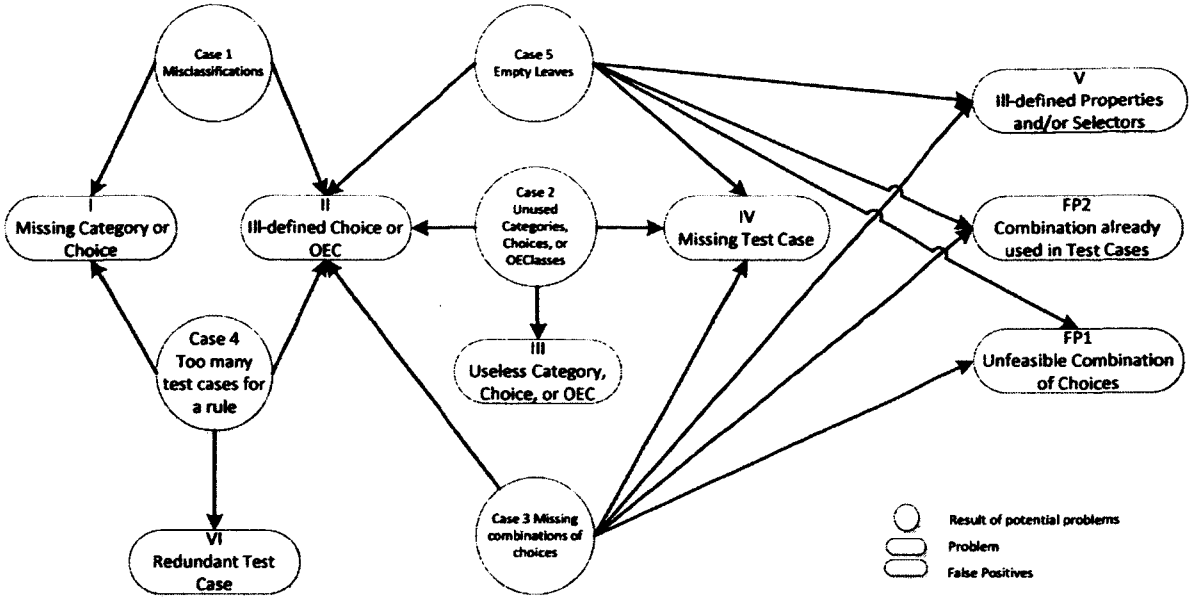

Figure 3 Potential problems identified through the analysis of results [12]

After the DT results are analyzed, if there are any potential problems remaining, the user can change the CP Specification or ATS data according to the problem identified. The process of refinement of test suites is summarized in Figure 4.

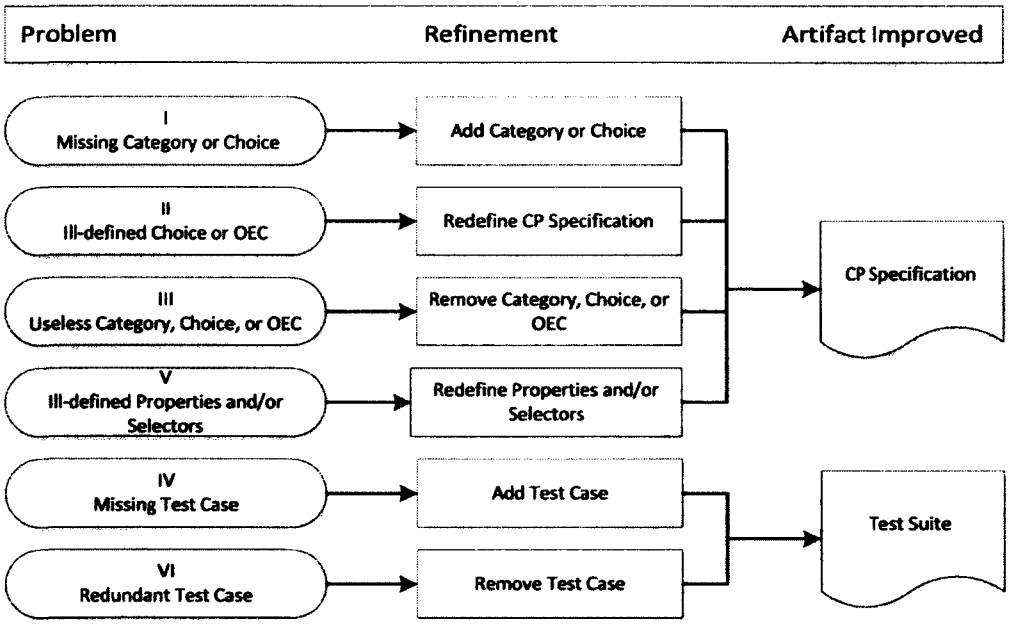

Figure 4 Mapping of problems, refinements and improved artifacts [12]

3.2 Shortfalls of Melba

As demonstrated, Melba has been used for refinement of test suites ([12], [16], [17], [18]). In the testing industry automation is very important for reduction in costs. By Generating test suites automatically the user does not have to build a test suite from scratch, which can be a long and tedious process. Melba has not been designed for generating test suites, which can be a very useful feature in the industry. This allows the user to avoid ad-hoc methods of generating test suites and also facilitates the generation of test suites by means of automation.

It is worth noting that since Melba already has capabilities for specifying and reasoning about a category partition specification; this makes Melba an immediate choice for creating tool support to automatically create tests for this black-box method.

3.3 Updated Melba

As mentioned in the previous section, we would like to have means of generating test suites automatically. It is important that the user avoids ad-hoc methods of generating test suites by using a selection criterion according to the testing needs. We have decided to use three selection criteria including Base Choice, Each Choice and Pair Wise.

We have added a new feature to Melba whereby the user provides the CP specification and selects a selection criterion and Melba will generate an ATS, i.e., a set of test frames, while accounting for constraints between chocies as well as constraints such as single and error on choices.

Since test frames are sets of choices and more specifically combinations of choices, identifying a set of test frames that satisfy a selection criterion such as Each choice and Pair wise is a combination testing strategy which can be implemented using mathematical objects such as orthogonal arrays or covering arrays [19]. Since we also account for constraints on and between choices, we are in fact deadling with covering arrays with forbidden edges/configurations [20].

In the following sections we will discuss covering arrays and why we have chosen Covering Arrays by Simulated Annealing (CASA) for the construction of test suites. Then we will discuss

the integration of Melba and CASA to obtain test suites. The next step is to discuss the functional and conceptual model of the new designed Melba.

3.3.1 Covering Arrays

Combinatorial interaction testing (CIT) is a black box sampling techniques. It is a cost effective technique for discovering interaction faults in highly configurable systems [21]. There are two types of algorithms explored in the literature to solve these combinatorial interaction testing problems including greedy techniques such as Automatic Efficient Test Case Generator (AETG) [22] or the In Parameter Order (IPO [G]) algorithm [23], and meta-heuristic algorithms such as simulated annealing ([24], [25]). Both these approaches construct solutions by making small, elementary decisions, but greedy algorithm's selection are permanent, whereas meta-heuristic search may revisit its choices [26]. Even though greedy algorithms are faster because they visit choices only once, meta-heuristic searches usually yield smaller sample sizes ([20, [19]).

A covering array (CA) is an array of N rows and k columns. Each element in the i^{th} column is a symbol chosen from a set of v_i symbols. For any choice of t columns, all possible sets of t symbols (t-sets) appear in at least one row and the t-sets are said to be covered [21]. For Pair-Wise testing, t would be 2 since we want to cover all 2-value set combinations. This value is known as the strength of testing. Constrained covering arrays (CCAs) incorporate constraint information into the greedy sampling algorithms without increasing computational cost.

Covering Arrays by Simulated Annealing (CASA) provides an effective and fast algorithm for obtaining covering arrays taking constraints into consideration, which is precisely our problem. In addition, CASA's C++ source code is readily available [21]. We have therefore decided to use it as the tool to create the test frames, and modified Melba so it can interact with CASA to generate test suites for the different selection criteria.

3.3.2 CASA inputs and output

CASA requires a specific format for construction of test suites [21]. In order to be able to integrate CASA and Melba we need to understand the inputs and outputs used by CASA. CASA

takes two files as input, including a model file and a constraint file, and returns a file with the covering array output. The model file has the following format:

[strength of testing(t)]\n
[number of options(k)]\n
[number of values in the first column(v_o)][more v_{is}]\n

As discussed in the previous Section 3.3.1, the strength would be 2 for Pair Wise test criterion and 1 for Each Choice criterion. We discuss later how we implemented the construction of a Base Choice adequate test suite.

The number of options maps to the number of categories of each parameter in the system and the number of values corresponds to the number of choices in each category. An example of a model file would be:

2
5
2 2 2 2 3

Where the first line is strength of two meaning Pair Wise coverage, the second line means we have five categories, and the last line indicates that the first category has two choices, the second category has two choices, and so on. With such a model file, CASA understands that there are eleven choices (2+2+2+2+3), which it refers to as symbols (in our context these are choices) and counts by starting at 0 (in our case we have symbols labeled from 0 to 10). To help understanding the contents of the constraint file, this model file corresponds to the categories and choices in Table 1.

Categories	1		2		3		4		5		
Labeled choices (symbols)	0	1	2	3	4	5	6	7	8	9	10

Table 1 Categories and Choices for CASA

Constraints are written as a conjunction of disjunctions over all symbols (i.e., 0 to 10 in our running example). The total number of disjunctive clauses is stated in the first line. Then the constraint file shows two more lines for each of the disjunctive clauses: the first indicates how many clauses is in the constraint, the second specifies the constraint. The format of the constraint is the following:

[number of disjunctive clauses] \n
[number of terms in a disjunctive clause] \n
[+ for plain, - for negated] [variable] [more plain or negated variables...] \n
[more disjunctive clauses...] \n

Below we continue our running example, with symbols numbered from 0 to 10. The first line states we have two constraints: lines 2 and 3 specify the first constraint while lines 4 and 5 specify the second constraint. The first constraint has two terms while the second has three. The second constraint reads -0-8, referring to symbols number 0 and 8, and specifying that the first symbol (numbered 0) cannot be combined with the ninth symbol (numbered 8). Because of the number of symbols (i.e., choices in categories) specified in the model file above (i.e., sequence 2 2 2 2 3), this means the first choice of the first category cannot be combined with the first of the three choices in the fifth category. The second constraint specifies that symbols numbered 4 and 7 can be combined but not with symbol numbered 3. In other words, the first choice of the third category can be combined with the second choice of the fourth category, but not with the second choice of the second category.

```
2
2
-0 -8
3
-3 +4 +7
```

The output then is constructed in the format:

[number of configurations (rows)] \n
[chosen value for the first option in the first configuration] [more chosen values for the first configuration] \n
[more configurations] \n

An example of the (incomplete) output is as follows:

```
27
0 2 4 6 10
0 2 4 7 9
0 2 5 6 10
...
```

Where 3 is the number of rows, i.e., the number of test frames, and each value in a row correspond to a symbol (choice). For instance, the first test frame (0 2 4 6 10) is the combination

of the first choice of the first category, the first choice of the second category, the first choice of the third category, the first choice of the fourth category and the third choice of the fifth category.

3.3.3 Melba and CASA working together

The CP Specification defined by the user using Melba is converted into the format required by CASA and these are written into two files. The constraints provided by the user have to be in conjunctive normal form so that they can be converted into a form that is used by CASA. This is not a strong restriction since Melba could be extended to include an algorithm that transforms all constraints in a conjunctive normal form is needed. After the conversion Melba executes CASA to obtain the resulting covering array.

It is worth noting at this stage that since this projet was a proof of concept of the use of combinatorial design algorithms to produce adequate test suites for selection criteria associated with Category Partition, we did not fully integrate CASA into Melba. Specifically, since Melba is written in Java and CASA is written in C++ we had two integration solutions:

1. Ensuring that Melba produces the right text files, calls the CASA program, and then collect the result in a text file produced by CASA;

2. Changing CASA so that Melba calls a dll, interacting with CASA through the computer memory instead of files.

Since the latter would have required changes to CASA and we were only looking at a proof of concept implementation, we implemented the first solution.

We also had to take specific care of error and single choices, which cannot be understood by CASA. We proceeded as follows in order to get error and single choices involved only once in the set of test frames: Figure 5. CASA is first run without the error/single choices: Melba creates a model file without those choices; we then run CASA again with the single choices (error choices are not yet considered). The two resulting covering arrays are saved by Melba. When constructing the adequate test suite, rows from the first covering array are all converted into test frames and only one row (the first row that contains a single choice) for each single choice is selected from the second covering array and is converted into a test frame. Lastly error choices

are added to the set of test frames: one test frame is created for each error choice and the test frame only contains this error choice. This ensures that error/single choices appear only once in the test suite. The error choices cannot easily be combined with other choices and this is why we chose to add the error choices as test frames at the end of the process.

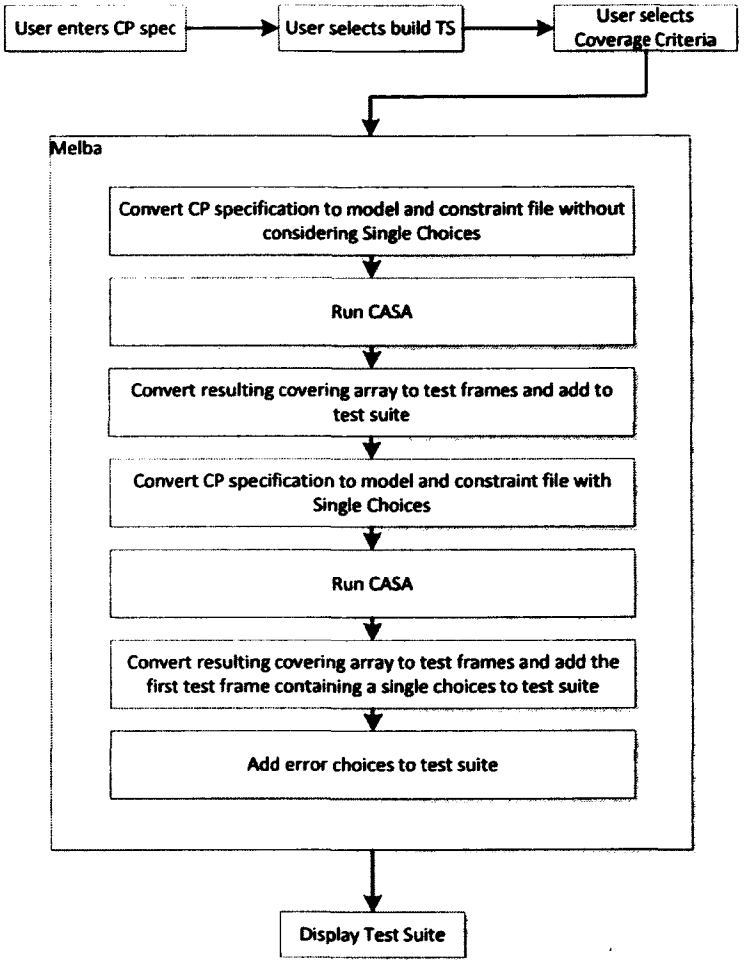

Figure 5 Melba New Functionality

We need to construct test suites for three criteria, including Base Choice, Each Choice and Pair Wise criteria. CASA is used for t-wise coverage only and does not have support for Base Choice. Therefore we have added the capability to CASA for constructing Base Choice test suites as well. We have constructed a simple algorithm and used the SAT solver in CASA to construct Base Choice test suite: the SAT solver is used to verify that the test frames generated by our algorithm satisfy the constraints on choices. Figure 6 shows this algorithm.

```
INIT bcArray to an empty list of choices (which is a Test Frame)
SET allCategories to all categories with choices (provided by user in
CPSpecification)
SET aCategory to a category from allCategories
SET allChoices to all choices in aCategory
SET aChoice to a choice from allChoices
INIT bc to a list of base choices from each category

FOR aCategory in allCategories
      INIT tempChoices to an empty list of choices
      FOR aChoice in aCategory
            COPY contents of bc to tempChoices
      ENDFOR
ENDFOR
RETURN bcArray
```

Figure 6 Base Choice Criterion Algorithm

We start by constructing a single test frame with only base choices and then we change one choice at a time, replacing the base by each of the other non-base choice of the category, to build other test frames. After building each test frame, we check with the SAT solver embedded in CASA if we have a valid test frame according to constraints. Then we add the test frame to the covering array to be returned by CASA.

For constructing covering array for Base Choice criterion, CASA assumes that the first choice in a given category is a base choice. Therefore we have rearranged the choices so that the base choice appears first. This will have no impact in the appearance of numbering when the test suite is constructed. This is done by Melba, and the user is not required to ensure the CP specification satfisfies this constraint.

3.3.4 Functional Description of Melba

The target user of Melba is a human actor interested in evaluating the quality of, and possibly improving, a given test suite, or constructing a test suite using a selection criterion. In both cases the user is responsible for defining the CP specification which includes defining parameters, categories and choices (including a base choice) for each category and defining constraints. If the user wants to evaluate an existing test suite, the user needs to define output equivalence classes for parameters as well as providing the test suite. The details of the format can be found in [12].

The Use Case diagram in Figure 7 shows Melba's use cases. UC1, UC2, UC3, UC4, and UC5 represent the main goals of the process of using Melba. UC6 is added (to the existing Melba) as a new functionality for making a test suite. UC1 is about loading the test suite in a TTCN-3 format and specifying the details of the CP Specification (input and output parameters, categories and choices) and output equivalence classes. UC2 is about the core process of Melba which is creating the ATS, running the Machine Learning algorithm and analyzing the results. UC3 and UC4 allow the user to modify the CP specification or the test suite (TS). UC5 is about saving the data and UC6 is about generating a test suite.

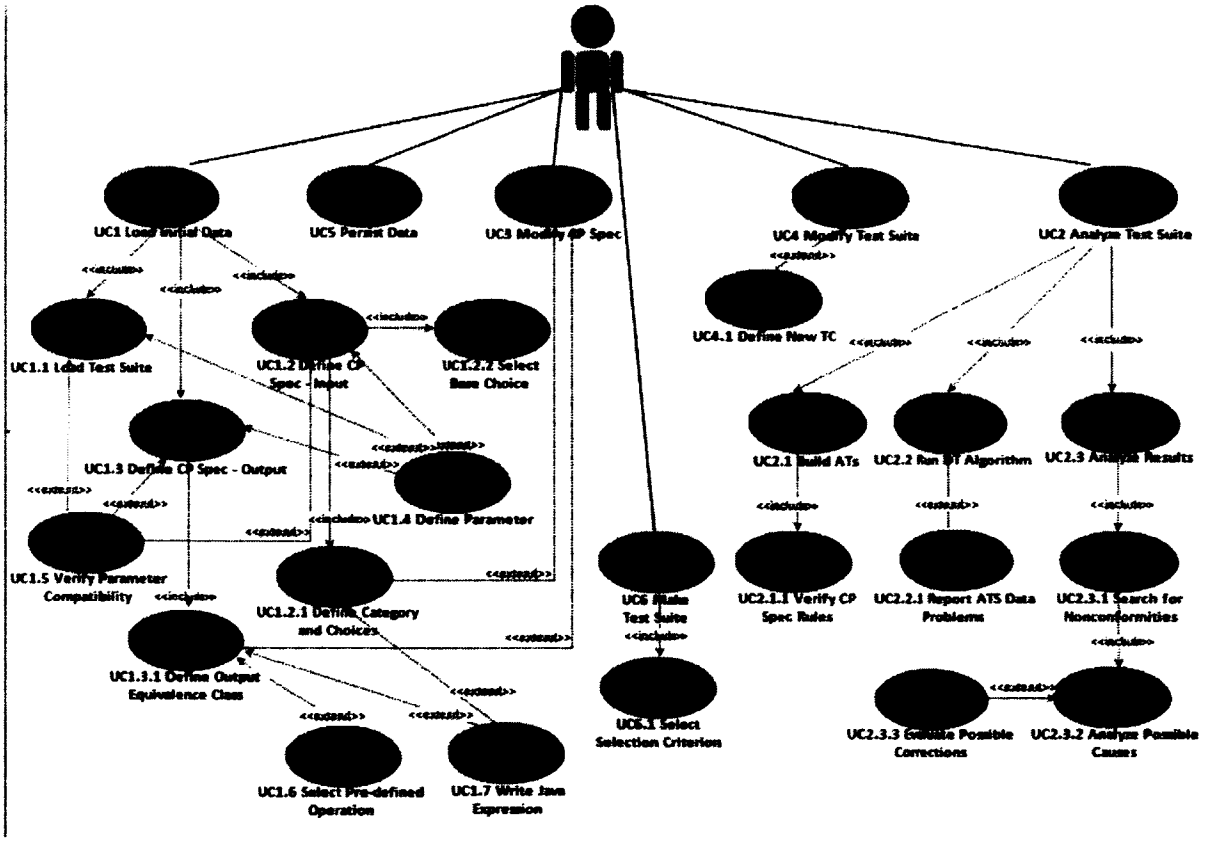

Figure 7 Melba Use Case Diagram

Initializing the data (UC1) includes loading the test suite (UC1.1) and defining the CP Specifications including input and output parameters (UC1.2 and UC1.3). In UC1.1 the user provides the test suite (TS) and the tool automatically identifies parameters and their types from this TTCN-3 file and loads them into the CP specification. Alternatively the user can define these parameters into the tool (UC4.1) and load the test suite later. The latter is what the user would do

if they were to use Melba to generate test suites. If the user defines parameters and loads the test suite later (for improvement of test suite), the compatibility of types is verified in UC1.5. The user is alerted if there are any compatibility issues and she can change the specification or test suite accordingly.

In UC1.2 the user defines categories and choices for each input parameter (UC1.2.1). The user can either use UC1.6 which allows them to select a pre-defined operation, or UC1.7 to write a JAVA expression. The user has to select a base choice (UC1.2.2), which is necessary for the Base Choice criterion used to build test suites. In UC1.3 the user defines output equivalence classes using UC1.3.1 (using either operations, UC1.6 or JAVA expressions, UC1.7) for output parameters. This step is only necessary if the user is using Melba for improving a test suite. If Melba is used for building test suites, this step is not necessary.

UC2 includes generating an abstract test suite (ATS) from a concrete TTCN-3 test suite (UC2.1), running a decision tree (DT) algorithm (UC2.2) and analyzing the results (UC2.3). UC2.1.1 identifies problems that are related to the CP specification as a result of generating the ATS and the user is prompted to correct the problem. UC2.2.1 captures and reports on problems that are found in the data that is to be used by the DT algorithm. In the case of UC2.3, if any potential problems identified at UC2.3.1, it is automatically analyzed for possible root causes (UC2.3.2) and presented to the user (UC2.3.3).

UC3 and UC4 allow the user to make modifications to the CP specification by redefining category and choices or output equivalence classes. UC5 allows the user to save the data.

UC6 allows the user to create a test suite for a selection criterion (UC6.1). This test suite is a set of test frames generated from the CP specification.

More detailed use case descriptions can be found in APPENDIX A USE CASE DESCRIPTIONS FOR MELBA.

3.3.5 Conceptual Model of Melba

In this section we will describe the new additions to Melba. The description for the previous classes can be found in [12]. The new class diagram, including previous and new classes, can be found in Figure 8.

We have added ten classes to the existing class diagram of Melba. These include `TestFrame`, `TSGenerator`, `SelectionCriterionAbstraction`, `CombinationImplementor`, `CASAImplementor`, `BaseChoiceCriterion`, `EachChoiceCriterion`, `PairWiseCriterion`, `CASA_v1_C_Adaptor` and `CASA_v1_c`. We have also added the enumeration type `SelectionCriterionID`.

`AbstractTC` is extended by `TestFrame` and a given `TestFrame` is an `AbstractTS` without an `OutputEquivalenceClass`. `CASAImplementor` uses `CPSpec` to construct many `TestFrames`. The `CASA_V1_C_Adaptor` is a `CASAImplementor` used to construct `TestFrames`. `CASA_v1_Adaptor` contains the functionality for conversion of a CP Specification to a format used by CASA and vice versa. `CASA_V1_c` is used for running CASA. `SelectionCriterionAbstraction` is an abstraction of the three selection criteria. `PairWiseCriterion`, `BaseChoiceCriterion` and `EachChoiceCriterion` classes inherit `SelectionCriterionAbstraction` used to select a specific algorithm for each criterion. `TSGenerator` generates a test suite using the criterion that is selected. The enumeration `SelectionCriterionID` is used by `CombinationImplementor` to identify which criterion is being selected for generation of a test suite and to construct the model file according to this criterion.

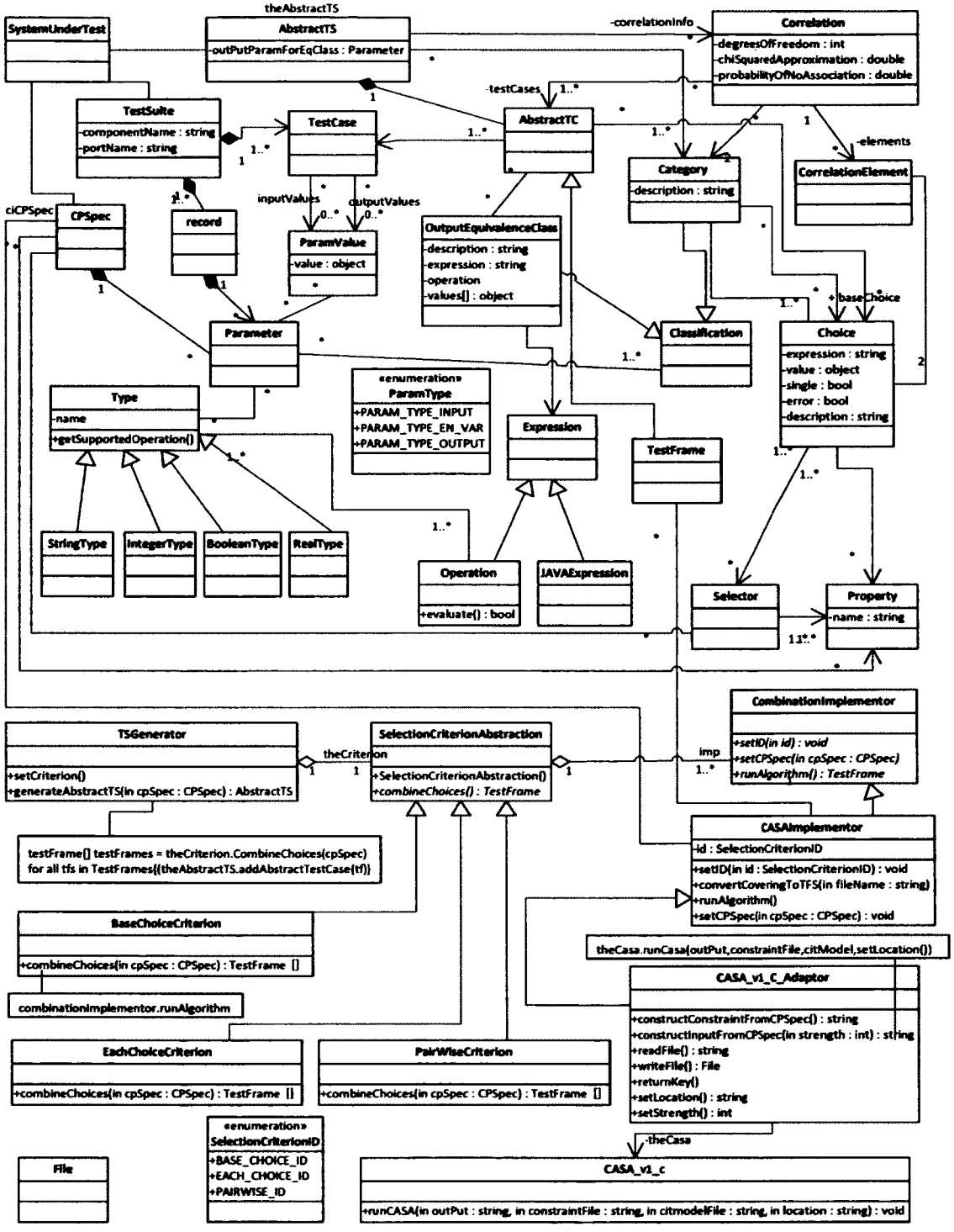

Figure 8 Melba New Class Diagram

3.3.6 User Interface of Melba

In this section we will show how a user can generate test suites using the new feature added to Melba. The user will select an existing test suite or construct a test new System. This is illustrated in Figure 9.

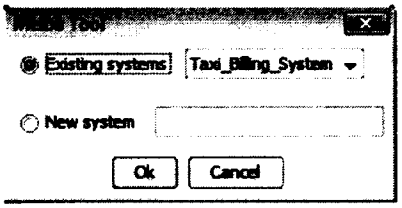

Figure 9 Choosing the SUT to work with [12]

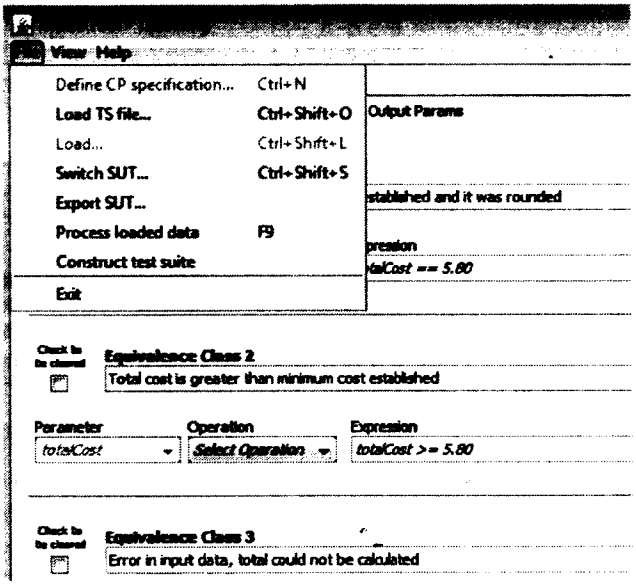

Figure 10 Choosing the SUT to work with [12]

Assuming that the user would like to start a new system, after entering the system name in Figure 9 the user can choose from one of the options in Figure 10. Assuming that the user has no information entered, they would have to define the CP specification using the form shown in Figure 11 and Figure 12. If they have already defined the CP specification previously and want to load it, they can simply use the load button, and select the file that was previously saved.

The user only needs to define the inputs to the system for construction of the test suite. Figure 11 shows the definition of parameters for the TaxiBilling Case Study. The user enters the name of each of the input parameters, selects the type (Input Parameter or Environment variable) and Data type (String, Boolean, Integer, Real) and adds these parameters to the system.

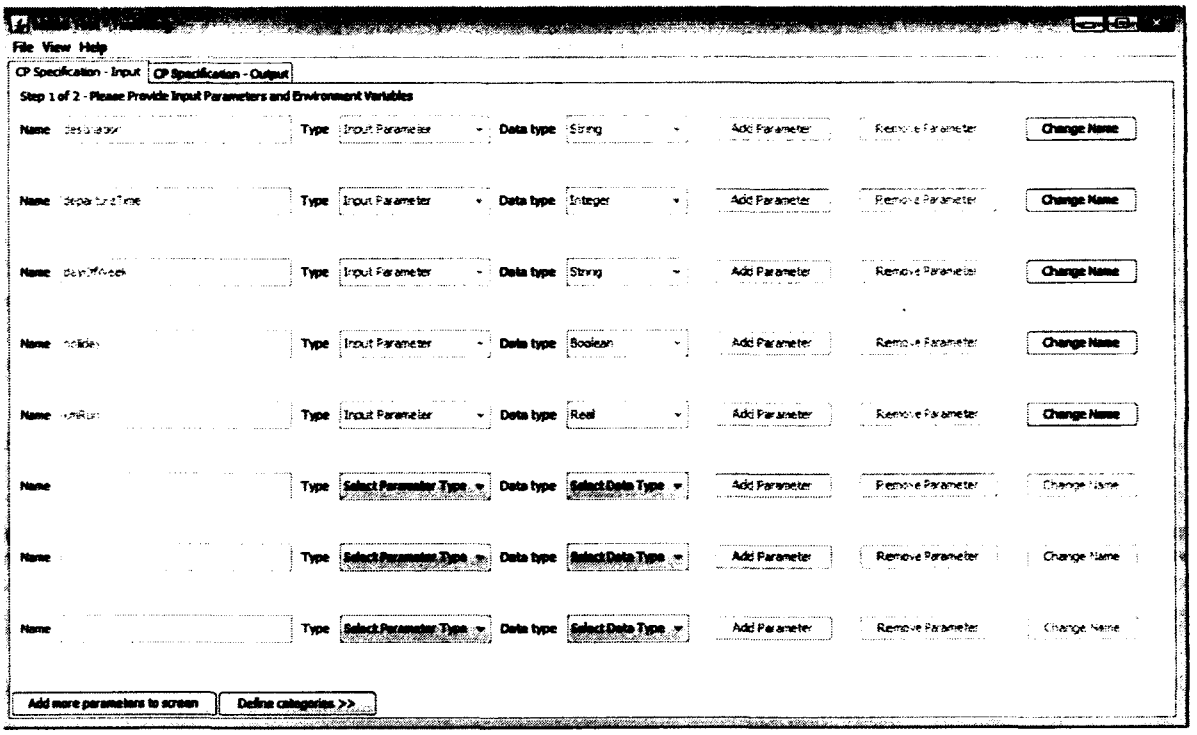

Figure 11 Melba CP Specification – Input Parameters

The user then needs to define the CP specification for the system. These include categories and their choices, the Base Choice for each category, single and error choices, and constraints. Figure 12 shows the specification of a base choice with a constraint for the taxiBilling system. As can be seen from the figure, the user has indicated a name for the Choice (i.e. Afternoon). The user can also select an operation, or enter an expression. This indicates valid input values for the choice. The properties section of the choice needs to be specified if another choice has constraints involving this choice. As can be seen from the figure the properties section is *isAfternoon*. This choice cannot be combined with another choice tagged `Sunday` and therefore a selector `if !isSunday` is indicated.

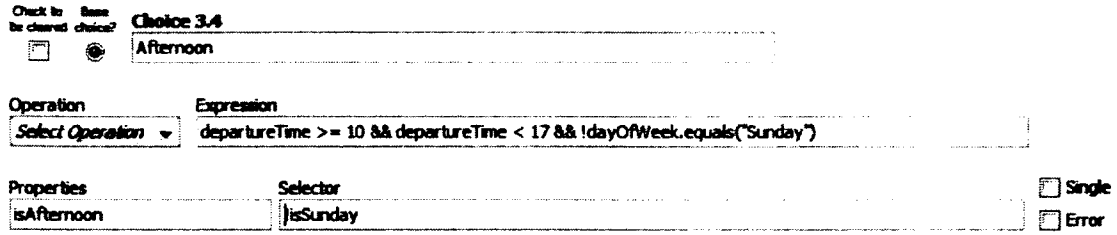

Figure 12 Demonstration of a base choice with constraint

The user can then select Construct test suite from the menu (Figure 10) to construct the test suites. Figure 13 shows the criteria that the user can select for constructing a test suite.

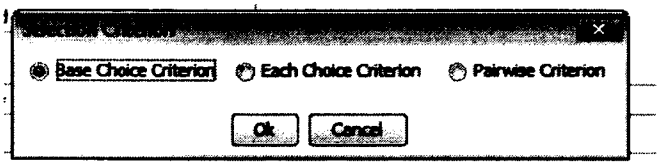

Figure 13 Selection of criterion

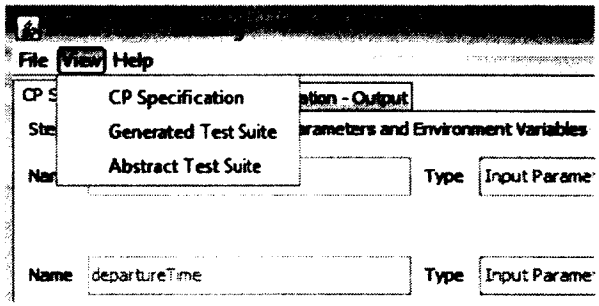

Figure 14 Viewing Generated Test Suite

The user can view the generated test suite by selecting Generated Test Suite from the menu as shown in Figure 14, which brings a new windows showing the list of test frames (Figure 15). Figure 15 shows 13 test frames: the first eight ones are combinations of six choices whereas the last five contain only one (error) choice.

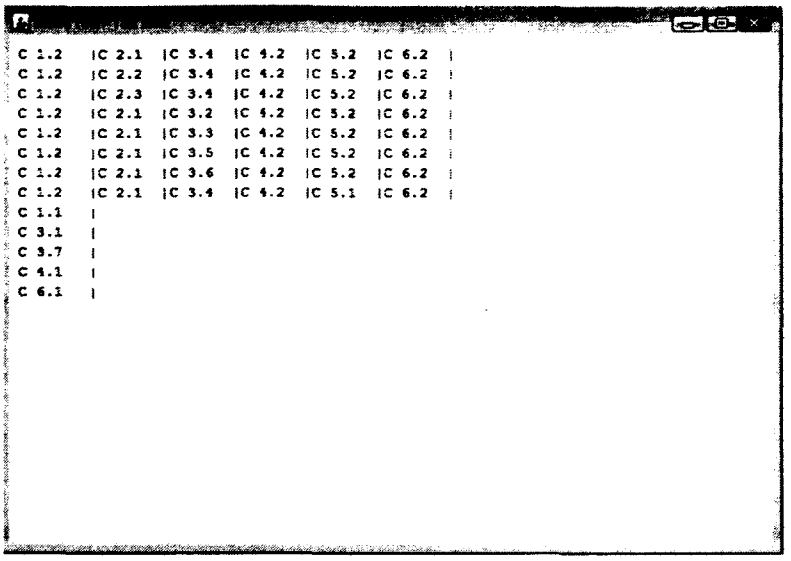

Figure 15 Generated Test Suite

4 DESIGN OF CASE STUDIES

The main goal of this thesis is to improve Melba so that we can build test suites using CP specifications and experiment with three selection criteria. These criteria include Base Choice, Each Choice and Pair Wise criteria. Recall that with Base Choice a base choice is selected for each of the categories. A first test frame is created by using all the base choices. Other test frames are then created from this first test frame by holding all but one base choice constant and using each non-base choice once for the one non-base choice. With Each Choice, each choice in every category must be in at least one test frame. And lastly with Pair Wise we exercise each pair of choices from different category at least once. [2]

For the purpose of reaching the goals we have constructed five case studies. In this section we will give a brief discussion of each case study system (section 4.1), the design of test cases from test frames (section 4.2), the generation of mutant programs (section 4.3), and the measurement of structural coverage for result analysis (section 4.4).

4.1 Case Study Description

The following sections include a brief description of each of the five case studies and the corresponding CP specifications can be found in Appendices B1, C1, D1, E1, and F1, respectively. These case studies are the *Triangle Problem* [27] (section 4.1.1), the *Next Date Problem* [27] (section 4.1.2), the *PackHexChar* problem [16] (section 4.1.3), the *OrdSet* Problem [28] (section 4.1.4), and the *Taxi Billing* problem [12] (section 4.1.5). The first two are specifications well-known to the testing community. PackHexChar is a simplified version of a commercial functionality. OrdSet is part of an academic tool for source code analysis: we only specified the method that computes the difference between two sets. The Taxi billing system is a simplified version of a real software specification that has been used as an exercise in a software V&V undergraduate course.

Although the specifications may appear simple and the corresponding implementations are not very long in terms of number of lines of code, these five case study systems and somewhat representative of cases where category partition would be used. Table 2 reports on characteristics of the category specifications. The *choices* column shows the total number of choices and the

average number of choices per category between parentheses. The *constraints* column shows the total number of constraints and the number of those which are error/single constraints between parentheses. The last column shows the number of lines of code that implement the functionality.

	parameters	categories	choices	constraints	LOC
Triangle	3	9	18 (2)	11 (6)	38
NextDate	3	4	17 (4)	9 (6)	186
PackHexChar	3	5	16 (3)	5 (5)	166
OrdSet	3	3	10 (3)	6 (2)	147
Taxi billing	5	5	19 (3)	9 (4)	325

Table 2 Characteristics of the five category partition specifications

4.1.1 Triangle Problem

The triangle problem is a classic problem to determine if three integers SA, SB and SC, each representing the length of a segment, represent a triangle or not, and to determine the type of the triangle (scalene, isosceles, equilateral). The output would be the type of the triangle, which are the following: `equilateral`, `isosceles`, `scalene`, or `notATriangle`.

As a geometry reminder, we note that three sides SA, SB, and SC denote a triangle if and only if the triangle inequalities hold, that is: SC < 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, and the triangle inequalities hold, then 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 is 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`.

The complete CP specifications for the triangle problem can be found in the Appendix B1.

4.1.2 NextDate Problem

The purpose of the next date problem is to find the day after an input date. There are three variables in this problem; these are year, month and day. The Georgian Calendar starts from 1582, therefore we chose this date as a starting date, and we make an assumption for the end date to be 2100. So the range of valid values would be [1582, 2100]. Anything 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.

The complete CP specification can be found in Appendix C1.

4.1.3 PackHexChar

The following description is taken from [16]. *PackHexChar* is a Java adaptation of the *sreadhex* procedure used in the *GhostScript* program, which processes the PostScript page-description language.

PackHexChar takes a string of characters representing hexadecimal digits (parameter S) and compacts the representation of the string in binary format (output), specifically as an array of Bytes: e.g., string "34AB", corresponding to binary values 0011, 0100, 1010, and 1011, is compacted into an array of two Byte values 00110100 and 10101011 (the binary representation of hexadecimal characters '3' and '4' are combined into the first byte value 00110100, and the same for characters 'A' and 'B' combined into the second byte value 10101011).

In the input string, characters other than hexadecimal ones are ignored. In addition to the array of Bytes, the program returns an integer value. If the input string contains an even number of hexadecimal characters, pairs of hexadecimal characters are compacted, the program returns the array of Bytes and the returned integer value equals to -1. In case the input string contains no hexadecimal character, no array of Bytes is returned and the returned integer value is also -1. If the input string contains an odd number of hexadecimal characters, an even number of characters is compacted, and the program returns the remaining hexadecimal character.

The user can decide to look at only a sub-string of the input string S, using the input parameter `rlen`: the `rlen` first characters of S are then analyzed. If `rlen` is not a legal value (negative or greater than S's length), the program returns value -2. The user can ask the program to append an hexadecimal character at the beginning of S. This is useful when a string is split and analyzed in pieces with repeated calls to *PackHexChar*: a call can return a trailing hexadecimal character, which has to be appended at the beginning of the string during the next call. This is done with input parameter `odd_digit`. An `odd_digit` value of -1 indicates that no character is to be appended. If `odd_digit` has an illegal value (strictly below -1 or not an hexadecimal value), the program returns -3.

The complete CP specification can be found in Appendix D1.

4.1.4 OrdSet

Class *OrdSet* represents a bounded, ordered set of integer values. [28] When an OrdSet is first created, its size gets initialized. The size of an OrdSet represents slots that can be used to add integers to the set. A size of 8 means that there is room for 8 integers, although the set may only contain 5 integers (i.e. 3 empty slots can be used for additional insertions). The size should be at least equal to the minimum set size (in our implementation, the minimum set size min_set_size is set to 4) and it should not exceed the maximum set size (set in the implementation to 16). The size of an OrdSet (i.e. _set_size) is always a multiple of min_set_size, i.e. 4, 8, 12 and 16.

The user can choose a size for the OrdSet by providing an integer value to one of the constructors, but the actual size gets initialized based on the constraints above. For instance, creating an OrdSet with:

a. Integer value of 2, its size would be initialized to 4
b. Integer value of 10, its size would be initialized to 12
c. Integer value of 11, its size would be initialized to 12
d. Integer value of 16, its size would be initialized to 16
e. Integer value of 32, its size would be initialized to 16

A single element can be inserted in the ordered set (add(int)); Removing a value from the set can be done with the remove(int) method.

An OrdSet can be resized when adding a new element while the set is full (i.e., the number of elements in the set has reached its _set_size). The number of resizes allowed is set to a constant (in our implementation, max_accepted_resizes is set to 2). Trying to resize the set over the max_accepted_resizes or for a size that exceeds the maximum set size (max_set_size = 16) would not be allowed; in that case, an overflow in the instance of OrdSet is detected and no more insertion or removal of elements is allowed on the ordered set. An attempt to add or remove an element from an ordered set after an overflow is detected would raise an OverflowException.

The basic set operation union is implemented for the ordered sets. A union of two OrdSet instances s1 and s2 would return an OrdSet s that contains all elements from both sets s1 and s2. A difference of two OrdSet instances of s1 and s2 would return an OrdSet that contains elements that are in s1 but not in s2. If there are no elements in any of the OrdSets the resulting OrdSet will be empty as well. Sets can have all elements in common only if the sizes of both sets are the same and the elements are the same.

In our case study we only worked on the difference function, which complete CP Specification of OrdSet can be found in Appendix E1.

4.1.5 Taxi Billing System

The case study problem is a Taxi Billing system based on the system operating for taxi services in Paris [12]. The purpose of this case study is to calculate the total cost of a trip made in Paris. The following rules are applicable:

1. Regardless of the calculated result by the system there is a minimum charge of € 5.80
2. The passenger does not pay the return trip for the Taxi.

3. The calculation is based on the destination area since the starting trip is from an Urban area. The starting point is downtown Paris and the destination can be downtown or different parts of the suburbs.
4. The departure time of the trip is used to calculate the charge applied.

Table 3 shows how to calculate the total charge for a trip. The initial charge is always added to the total amount to pay at the start of any trip.

APPLICABLE CHARGES								
Initial Charge: € 5.80	Charges per Kilometre	A						
		0.86 €/km			1.12 €/km		1.35 €/km	
Time zone	Monday to Saturday					Sunday and Holiday Sunday		Holidays
	0 to 7am	7am to 10am	10am to 5pm	5pm to 7pm	7pm to 0	0 to 7am	7am to 0	any time
Urban Area Paris, *including peripheral* road			A					
Suburban Area End of Paris Taxi's area and airport services for Orly, Roissy and Du Parc, and Villepinte expositions								
Beyond Suburban Area								

Table 3 Applicable Charge for the Taxi Billing System

The expected input data for computing the total amount to pay for the trip is: Destination area, Departure time, Day of the week, Whether it is a holiday or not, Total kilometres run. The expected output data is: Total cost of the trip, Charge applied (A, B, or C).

Error situations include: Missing or unknown destination area, Missing or unknown day of the week, Time value less than 0 or greater than 23, Kilometres run less than or equal to 0. In the event of any of those error situations, the system reports -1 as the total cost of the trip and the charge applied is empty.

The complete CP specification of Taxi Billing can be found in Appendix F1.

4.2 Design of Test Cases: i.e., from test frames to test cases

As described in Section 3.3, a test case is created from each test frame produced by Melba, which means we need to select test inputs that correspond to the combination of choices in each test frame. The selection of inputs was done manually by choosing a value for choices in a test frame. In this section we will describe the process of selection of inputs for each test of the case studies.

There are nine test frames generated for the Base Choice test suite, seven test frames for the Each Choice test suite and ten for the Pair Wise test suite for the Triangle case study. Test input values for test frames with no error choice were chosen at random according to the CP specification, i.e., within the range specified by the choices while satisfying the constraints. There are three error choices in the CP specification, which resulted in three test frames in each test suite. Each error choice specifies an illegal (strinctly negative) value for each of the three inputs. The value of the error choice was chosen as 0 and the values for the other two choices in the test frames where chosen randomly in their legal range. These test cases are the same for all test frames as to not give more advantage or disadvantage to one test suite over another. In other words, if two test suites have a common test frame, the two test suites have a common test case. Since the set of test frames have differences (because of the selection criteria), the adequate test suites have different test cases. After the selection of inputs for test cases the original un-mutated program was run to obtain output values and these values where then compared against values returned by mutants (section 4.3 discusses the construction of mutants): a difference between the two values for a given test case for a mutant resulted in the mutant being killed. The test frames and the corresponding test case input values and expected output values for the Base Choice, Each Choice and Pair Wise test suites are shown in Appendix: Table 45, Table 46 and Table 47, respectively.

There were 14 test frames generated for the Base Choice test suite, ten for the Each Choice test suite, and 19 for the Pair Wise test suite for the NextDate case study. The values for the year and month were chosen at random (within the range specified by the choices, while satisftying constraints) for this case study. The value of day however was chosen so that there are some that are at the boundary of the range specified by the choices. An example is the choice day>0 && day<=28 which led us to sometimes select value 28. Boundary values such as 28 were not

specified in specific choices, which might indicate a different, more detailed CP specification could have been used. The analysis of results might tell us whether a more detailed CP specification would have been better. The values for error choices where chosen randomly in the error range and the values for other choices in the test frame were chosen randomly in their legal range. Values for test frames with error choices are the same for all test frames as to not give more advantage or disadvantage to one test suite over another. After the selection of inputs for test cases the original un-mutated program was run to obtain output values. The values where then collected to be compared against mutants. The test frames and the corresponding test case input values and expected output values for the Base Choice, Each Choice and Pair Wise test suites are shown in Appendix: Table 49, Table 50, and Table 51, respectively.

There were 12 test frames generated for the Base Choice test suite, seven for the Each Choice test suite and 13 for the Pair Wise test suite for the PackHexChar case study. The values of the input string, `rlen` and `odd_digit` where chosen randomly from the ranges specified by the choices for each of the test suites. The length of all input strings with even length is the same (i.e. string length of four charecters) in the Base Choice and Each Choice test suite. The Pair Wise test suite however contains two test cases with a length of input string equal to six. These values are chosen by chance, that is they were selected this way and are not due to the CP specification of the test frame specifications. The length of the input strings with odd length is the same for all test cases (value equal to 3) in all test suites except for one test case in the Pair Wise test suite where the length is five. Again, these values are chosen by chance. The values for error choices where chosen randomly in the error range and the values for other choices in the test frame were chosen randomly in their legal range. Since test frames involving error choice only contain that error choice, the three test suites contain the same test frames for error choices. In addition the same values were selected for those test frames, resulting in the same test cases being part of the three test suites. The test frames and the corresponding test case input values and expected output values for the Base Choice, Each Choice and Pair Wise test suites are shown in Appendix: Table 53, Table 54, and Table 55, respectively.

There were seven test frames generated for the Base Choice test suite, five for the Each Choice test suite and eight for the Pair Wise test suite for the OrdSet case study. The size of an OrdSet and the values for the elements in the sets to compute the difference were chosen at

random, since there is no indication in the CP Specification for the content of each set. When creating the sets to be used in the difference function, once we knew which elements to add in the sets, we added them to the sets in increasing order of their values: for instance if we needed to create a set of integers with values 1, 4, and 10, we created an empty set and added 1 and then 4 and then 10. There was one exception: the randomly chosen values for one of the two input sets in one test case for the Each Choice test suite were added in a random order: for instance we would add 4 and then 1 and the 10. The values for the sets where chosen randomly for all test cases in test suites. There are two test cases however where the values chosen for a test case in the Base Choice and Pair Wise test suites are consecutive: e.g., 1, 2, 3. The values for error choices where chosen randomly in the error range and the values for other choices in the test frame were chosen randomly in their legal range. These test cases are the same for all test frames as to not give more advantage or disadvantage to one test suite over another. The test frames and the corresponding test case input values and expected output values for the Base Choice, Each Choice and Pair Wise test suites are shown in Appendix: Table 57, Table 58, and Table 59, respectively.

There are 12 test frames generated for the Base Choice test suite, 11 for the Each Choice test suite and 23 for the Pair Wise test suite for the TaxiBilling case study. If the value of `dayOfWeek` was `weekDay`, we started by choosing `Monday` and increasing the `dayWeek` for every other test case containing `weekDay` (i.e. `Tuesday`, `Wednesday`, etc). If there were several test frames involving the `destinationArea` as `Urban`, we chose `Paris` for the first test frame and then use all other values if there were more than one test frame involving `Urban` (i.e. `PeripheralRoad`). For the `timeOfDeparture`, we chose a random value inside the range for all test cases. The value 7 for time of day, which is a boundary value of the range specified by a choice, was chosen for one test case in the Each Choice test suite, and three test cases for the Pair Wise test suite. The value for `totalKm` was chosen randomly for all test cases.The values for error choices where chosen randomly in the error range and the values for other choices in the test frame were chosen randomly in their legal range. These test cases are the same for all test frames as to not give more advantage or disadvantage to one test suite over another. The test frames and the corresponding test case input values and expected output values for the Base Choice, Each Choice and Pair Wise test suites are shown in Appendix: Table 61, Table 62, and Table 63, respectively.

4.3 Mutants

In order to evaluate the effectiveness of test cases and therefore test suites at detecting faults, mutation analysis is used. As mentioned in [5] mutation testing is a fault-based testing technique that measures the effectiveness of test cases. These simple faults which are small changes to the program by applying *mutation operators* are called *mutations* or *mutants*. *Mutation operators* are small syntactic changes to the program.

MuJava is a Java-based tool that automatically generates mutants given a set of mutation operators, run the mutants against a set of test cases and generates a report on the mutation score of the test cases and test suite [29]. MuJava uses two types *of mutation operators*, called *class level* and *method level* operators. The *class level operators* address OO programming faults and ensure all OO features are tested. The two categories of OO features are inheritance and polymorphism. There is another category that is specific to Java programming. One of the few features includes `this` keyword which is specific to Java. The summary of the features in each category along with the abbreviation used in MuJava for the class level operators is summarized in

Language Feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call to a parent's constructor deletion
Polymorphism	PNC	New mothod call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter cariable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-Specific Features	JTI	this keyword insersion
	JTD	this keyword deletion
	JSI	static modifier insersion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and contet assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Figure 16.

Language Feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call to a parent's constructor deletion
Polymorphism	PNC	New mothod call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter cariable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-Specific Features	JTI	this keyword insersion
	JTD	this keyword deletion
	JSI	static modifier insersion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and contet assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Figure 16 Mutation Operators for Inter-Class Testing [5]

Method level operators as the name suggests, introduce operators for the body of methods of a class. There are six types of operators which are Arithmetic, relational, conditional, shift, logical and assignment operators. The mutations remove, replace or add these operations in the code. The summary and abbreviation used for these mutants is in Figure 17.

Please refer to [30] and [31] for more details on Class level and Method level operators.

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Figure 17 Method-level Mutation Operators for Java [31]

If a mutant is detected we say that it is *killed*, and if it is not detected we say that it is *alive*. After running test cases against mutants and determining the result, we need to analyze the results and determine why each mutant is killed or alive.

A mutant can remain alive because of the following reasons:

1. The mutant is equivalent to the non-mutated program, meaning that it does not change the functionality of the code and no test input can distinguish the mutant from the original program.
2. The mutant is not equivalent:
    a. The mutated statement is not executed (lack of code coverage) by the test suite.
    b. The mutated statement is executed (there is code coverage) but the test case did not led to mutant detection. This can have two causes with respect to the definition of test inputs:
        1) A different selection of test inputs from the CP specification could have led to killing the mutant.

2) No input selection from the CP specification could have killed the mutant, which suggests that the CP specification is not specific enough for the needs of constructing the test suite.

We have used MuClipse which is a MuJava plug-in for Eclipse. This tool allows the generation of mutants and running of mutants against test cases, which are written in JUnit. Since this tool allows running of J-Unit tests and is an easy to use interface, we have chosen to use this tool for mutation analysis. MuClipse generated mutants for the five case study systems as reported in Table 4.

	Triangle	NextDate	PackHexChar	OrdSet	TaxiBilling
Number of mutants	236	109	579	460	316

Table 4 Summary of mutants for the five case studies

4.4 Code Coverage

Code Cover [32] is a tool used for determining the code coverage for a given set of test cases. The coverage includes statement, loop, branch and term. If each statement of the program code is covered by a test suite, the statement coverage is 100%. If there is any branch in the code, and all branches (e.g. if-else branch) are executed, there is 100% branch coverage. Loop coverage helps careful testing of loops. The tool indicates whether a loop was not executed, executed once or more than once. Term coverage is a Modified condition/decision coverage (MC/DC). The following needs to be true for term coverage:

1. *Every decision in the program has taken all possible outcomes at least once*
2. *Every condition in a decision in the program has taken all possible outcomes at least once*
3. *Every point of entry and exit in the program has been invoked at least once*
4. *Every condition in a decision has been shown to independently affect that decision's outcome* [33] [34]

We used Code Cover in our case studies to determine which parts of the code is covered by each test suite. This tool will allow us to better explain that some mutants are not killed due to lack of code coverage.

5 Results

After the generation of test suites for each of the coverage criteria and generation of mutants for each test suite, test cases were constructed. These test cases were then run against the mutants created in each case study. In the following sections we will discuss the individual results for each case study: Triangle (section 5.1), NextDate (section 5.2), PackHexChar (section 5.3), OrdSet (section 5.4), TaxiBilling (section 5.5). We then summarize the results in section 5.6.

5.1 Triangle Case Study

In this section, we provide a structured, general discussion of the results. More specific, detailed results can be found in Appendix B3. The count of mutants killed (or not) by the different test suites is illustrated by the Venn diagram of Figure 18.

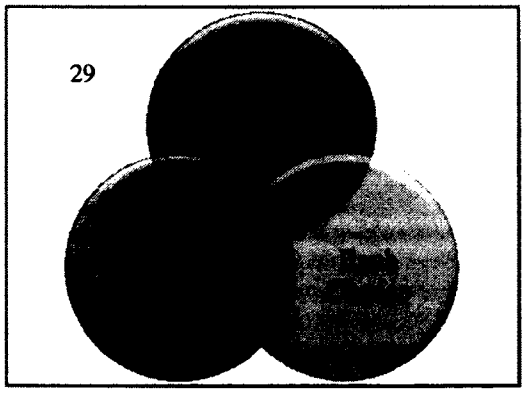

Figure 18 Triangle Case Study: Venn diagram illustrating the counts of mutants killed by the different test suites

There are 29 mutants that are not killed by any of the test suites. 19 of these mutants are not killed because they are equivalent. These mutants cannot be killed regardless of the CP specification and input values of test suites. Three mutants can be killed by adding test cases that have values less than zero. Since a value of less than or equal to zero was an error choice in the CP specification, the value zero was chosen once for each test frame. By splitting these choices (choices 1.2, 4.2, and 7.2) into two choices where one choice is for a parameter value equal to zero and the other is for a parameter value strictly less than zero, these mutants can be killed.

By adding a test case where SA is equal to SB+SC and SB equals to SA+SC, we can kill two other mutants. Test frames in all test suites allow for such inputs but do not enforce the selection of such inputs: one would need to split the choice SA>=SB+SC into two choices (SA>SB+SC and SA=SB+SC). Adding a test case where the value of SA is not less than SB and is scalene, we can kill one more mutant. The inputs can be changed to accommodate such a test case with the current test suites but these values are not enforced. Therefore adding a choice to the CP specification where the values of SA and SB are compared would ensure this mutant is killed. Two mutants can be killed by adding one test case where the value of SA-1 is equal to SC and value of SA+1 is equal to SC. Two last mutants can be killed by adding one test case where SA is equal to SC and is isosceles. It seems difficult to imagine a rationale for changing the CP specification to ensure these last five mutants would be killed: for instance, given the problem statement of identifying the kind of a triangle, why would one think of comparing the values of a-1 and c? Except perhaps if we consider that these inputs allow us to have test cases where the values lead to an iscosceles triangle but are very close to an equilateral triangle.

There are 52 mutants that are not killed by the Each Choice test suite. Since the Each Choice test suite has to cover each choice once, there are no test frames that lead to an output of scalene and isosceles. This is only due to a lack of chance that resulted from the way we selected input values. A different test input selection mechanism could produce such inputs, kill those mutants and increase the mutation rate. Alternativelly, another set of test frames produced from the same CP specification might lead to a similar outcome. The Base Choice criterion also has some limitations in terms of test frame generation. There are no way the Base Choice criterion can lead to a test case with inputs specifying an equilateral triangle because of the criterion and our CP specification: base choices specify that every pair of sides have different values (e.g., SA!=SB) so the one test case with all base choices is a scalene triangle (assuming inequalities hold) and the other test cases where the inequalities hold have one and only one pair of sides with equal values (isosceles triangle) and no test case involves all three non-base choices where every pair of side values are equal. This leads to eleven mutants not to be killed. Three mutants can be killed by changing the input values of the test frames. We conclude that adding choices to compare values of parameters individually (i.e., SA<SB, SA=SB, SA>SB, etc.) we can

have a better CP Specification and more mutants killed. We can maximize the number of mutants killed by choosing better values for inputs as well.

A better test suite would be one that includes inputs, not only at the boundaries of their own range, but at the boundaries of the domains they define with respect to outputs. This means adding for example an input that is `isosceles` but very close to an `equilateral` triangle.

All mutants killed by the Each Choice and Base Choice test suites are killed by the Pair Wise test suite, with one exception. This mutant is killed by the Base Choice test suite but not the Pair Wise test suite. Changing the values of the inputs will allow the Pair Wise test suite to kill this mutant, however other mutants will still remain alive. Only a change to the CP specification, whereby choices would compare values of parameters individually, would ensure this mutant be killed.

Although the Base Choice test suite has a better overall mutation score, seven of the mutants that it leaves alive are killed by the Each Choice test suite. The nature of the Base Choice criterion does not allow its test suite to have a test case with an `equilateral` triangle as discussed previously: base choices for Categories Cat2, Cat5 and Cat8 (`[ch2.2] SA != SB`, `[ch5.2] SB != SC and [ch8.2] SC != SA`) require that two of the input parameters are not equal to each other; Since at least two of these choices reside in a test frame, we cannot have an `equilateral` triangle. Since the Each Choice test suite contains a test case with an `equilateral` triangle, it is able to kill these mutants.

As can be seen from Table 5, the Pair Wise (PW) test suite gives the highest percentage of code coverage, followed by the Base Choice (BC) and the Each Choice (EC) test suites.

Method	Statement			Branch			Loop			Term		
	BC	EC	PW	BC	EC	PW	BC	EC	PW	BC	EC	PW
Triangle	0	0	0									
Triangle	100	100	100									
hasValidValues	100	100	100	100	100	100				100	100	100
triangleType	80	60	100	62.5	87.5	100				83.3	55.6	100

Table 5 Triangle Case Study: Code Coverage

In terms of the best test suite to use in the triangle case study, Pair Wise is the best criterion to use. This test suite has a higher percentage of mutants killed and more coverage. In terms of

the number of test frames, the Pair Wise test suite has more test cases than the Base Choice and the Each Choice test suites.

The overall statistics of the mutants is shown in Table 6.

Criterion	Number of test cases	Overall mutation score
Base Choice	9	Live mutants: 43 Killed mutants: 193 Mutation Score: 81.0%
Each Choice	7	Live mutants: 80 Killed mutants: 156 Mutation Score: 66.0%
Pair Wise	10	Live mutants: 35 Killed mutants: 201 Mutation Score: 85.0%

Table 6 Triangle Case Study: Overall Statistics

5.2 NextDate Case Study

In this section, we provide a structured, general discussion of the results for the NextDate case study. More specific, detailed results can be found in Appendix C3. The count of mutants killed (or not) by the different test suites is illustrated by the Venn diagram of Figure 19.

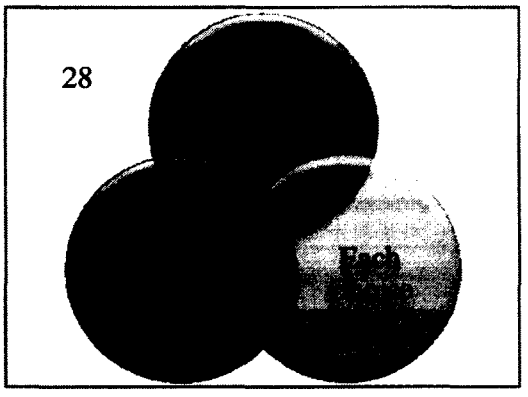

Figure 19 NextDate Case Study: Venn diagram illustrating the counts of mutants killed by the different test suites

40 mutants are killed by all test suites. 28 mutants are not killed by any test suite. 14 of these mutants are equivalent. Seven of the 14 remaining mutants not killed by any test suite make changes to a part of the code that is triggered when the month equals to February and the day is greater than 29 (i.e., an erroneous day for the month of February). This combination is infeasible according to the CP specification and therefore no test case can be constructed to kill these

mutants. Note that these inputs could be selected in a test case: we simply mean that such a test case cannot result from using our CP specification.

Two mutants require a test frame with a choice that leads to the next date's year to be invalid (i.e., the next date's year of input 31-12-2099 is 2100). The Pair Wise test suite can kill these mutants because it has to cover the combination of month equal to December and day equal to 31. However, it does not because value 2099 is not the only value specified by a choice: rather choice 1.2 specifies legal year values in the range [1582, 2100] and therefore the test frame that involves choices 1.2, 3.4 (December) and 4.6 (31) lead to selecting any value of 31-12-1582, 31-12-1583, ... 31-12-2099, and 31-12-2100 (each being equivalent as they satisfy the conditions of the test frame) and we did not select 31-12-2099. The Each Choice test suite that we have constructed contains such a test frame as well (likely by chance) but the test suite leaves the mutant alive for the same reason. The Base Choice test suite however cannot cover this combination because values 31 (for day) and 12 (for month) are not base: the Base Choice test frames do not each contain two non-base choices; and therefore is not able to kill these two mutants.

Four mutants require a test case with February 28th on a leap year. No test suite requires 3-tuples of choices to be exercised so the Each Choice and Pair Wise test suites may trigger this situation by chance only: in our case they do not. The Base Choice test suite cannot because February and isLeapYear are non-base. One mutant requires month to be February and day to be strictly smaller than 28. The range of values specified by the choice for the day is [1, 28]. The Pair Wise test suite does contain a test frame with February and a day in range [1, 28] but value 28 was chosen (which is in the range). If the value is changed from 28 to a smaller value in the range, this mutant will be killed at the expense of two other mutants becoming alive. Therefore adding 28 as a separate choice to the CP specification would allow all these mutants to be killed: choice 4.1 (day in [1, 28]) would be split into two choices, day in [1, 28[and day is 28. The Each Choice test suite may be able to kill this mutant depending on the test frame built for the value 28 for parameter day. The Base Choice test suite would still not be able to kill this mutant because neither February or 28 would be a base choice and would not appear in the same test frame.

Although the Pair Wise test suite gives a better overall mutation score than the other criteria, there are four mutants that are not killed by this test suite but are killed by the Base Choice test suite. These mutants are not killed simply because of our selection of input values for test frames, which resulted in different boundaries being exercised in the test suites. We identified that adding another choice for the boundary condition "day = 28" would allow the Pair Wise test suite to kill these mutants.

In the nextDate case study, the Each Choice test suite appears to perform better than the Base Choice test suite. To try to explain this, we look at the 22 mutants killed by the Each Choice test suite but not by the Base Choice test suite. These mutants are not killed due to the absence of a test case that covers some parts of the code. These parts of the code would be covered if there was a choice with month equal to 4, 6, 9 or 11 and day equal to 30, month equal to 12 and day equal to 30 and lastly month equal to 2 and day equal to 28 in a leap year. A given test frame in the Base Choice test suite only has one choice that is non-base choice. All the above-mentioned test cases require a test frame with more than one non-base choice or no base choice at all. Therefore no Base Choice test suite would not be able to kill these mutants because of the limitation of the criterion on the number of combinations of choices it triggers. Plus, it is difficult to imagine a CP specification that would lead to these conbinations being used in test frames produced by the Base Choice test suite.

There are 12 mutants that are not killed by the Each Choice test suite but are killed by the Base Choice and Pair Wise test suites. Four of these mutants require a month with 30 days and the day to be less than 30. Four other mutants require the month to be 12 and the day to be less than 31, and another four mutants require a month that has 31 days and day to equal 31. None of these mutants can be killed because there is no test frames with these combinations in the Each Choice test suite. However, these combinations could in theory be part of an Each Choice set of adequate test frames: it happens they are not in the one we created with CASA.

The way the Each Choice test suite is created offers more chances for triggering combinations of choices than any Base Choice test suite. Therefore in this case study the Each Choice criterion offers more random test frames where the combinations can kill more mutants. From Table 7 we can see that the Each Choice test suite has more code coverage than the Base Choice test suite.

Method	Statement			Branch			Loop			Term		
	BC	EC	PW	BC	EC	PW	BC	EC	PW	BC	EC	PW
getNextDate	59.1	63.6	77.3	51.2	56.1	78	-	-	-	50	58.8	76.5
NextDate	0	0	0	-	-	-	-	-	-	-	-	-
NextDate	100	100	100	-	-	-	-	-	-	-	-	-

Table 7 NextDate Case Study: Code Coverage

Since the Pair Wise criterion ensures that all pairs are covered, the Pair Wise test suite has better mutation score than the Base Choice and Each Choice test suites. However the specifics of this NextDate case study suggest that 3-tuples of choices could be interesting (e.g., ensuring we obtain a test case like February 28 on a leap year requires that we ensure we combine three choices. We believe this is why the pair Wise test suite only reaches a mutation score of 81.8%. Adding the test case above would kill four more mutants.

The overall mutation score of the Pair Wise test suite is the highest with 81.8% and ten test cases, the Each Choice test suite is next with 62.6% mutation score and seven test cases and the Base Choice test suite has the lowest overall mutation score of 56.5% with nine test cases. The summary can be found in Table 8.

Criterion	Number of test cases	Overall mutation score	Overall mutation score without equivalent mutants
Base Choice	9	Live: 57 Killed: 56 Score: 49.5	Live: 43 Killed: 56 Score: 56.5
Each Choice	7	Live: 51 Killed: 62 Score: 53.9	Live: 37 Killed: 62 Score: 62.6
Pair Wise	10	Live: 32 Killed: 81 Score: 70.8	Live: 18 Killed: 81 Score: 81.8

Table 8 NextDate Case Study: Overall Statistics

5.3 PackHexChar Case Study

In this section, we provide a structured, general discussion of the results. More specific, detailed results can be found in Appendix D3.

379 mutants are killed by all test suites and 109 mutants are not killed by any test suite. 41 of those 109 mutants are equivalent. The count of mutants killed (or not) by the different test suites is illustrated by the Venn diagram of Figure 20.

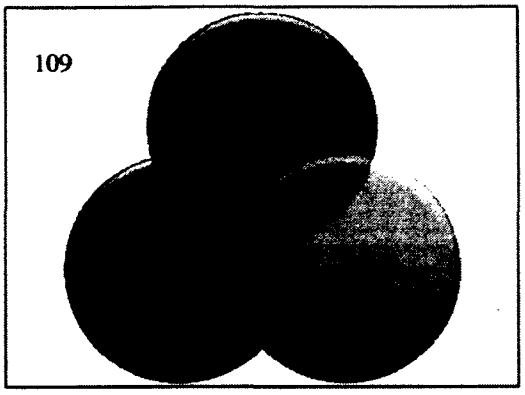

Figure 20: PackHexChar Case Study: Venn diagram illustrating the counts of mutants killed by the different test suites

Nine mutants are not equivalent but are not killed by any of the test suites. These mutants involve making changes to the `equals` function, which compares two `PackHexChar` objects The CP specification and therefore the test frames and test cases focused on the functionalities offered by a `PackHexChar` object, which consists in compacting information contained into integer values. The comparison of two `PackHexChar` objects was not part of the initial functionalities being tested. Therefore no test case was generated for this functionality and mutants in the `equals` function were not killed.

18 mutants are not killed by the Base Choice test suite. Since the Base Choice test suite does not contain a test case where the value of the input string has characters at the boundary of their range of values (i.e., F or f), four mutants are not killed. According to the CP specification the content of the input string is either hexadecimal, no hexadecimal or mixed characters. If there was an indication of the actual content for the string with boundaries (i.e., character in input string <A, A-F and >F), then these mutants would be killed. These mutants are killed by the Each Choice and Pair Wise test suites because the inputs where selected such that there is at least one test case with the input string containing f or F.

Two mutants are not killed by the Base Choice test suite because there is no test case where `rlen` is even and `odd_digit` is -1. Since the specification does not mention that an `rlen` value being odd or even would result in functionally different behaviors, no such category (and therefore choices) were defined for `rlen` in the CP specification. Our test input selection was

therefore not forced to exercise odd and even values for `rlen`, and it happened that all our test cases (in all our test suites) use odd values: no test case exercises an even value for `rlen`.

The Each Choice test suite does not contain any test case where `odd_digit` is not equal to -1 and `rlen` has a valid value. This is due to the limitation of the Each Choice criterion: it does not enforce this combination to be exercised, and therefore (because of a lack of chance) it is not. These choices do not appear in the same test frame and therefore are not covered by any test case. As a result, 59 mutants are not killed.

Two other mutants are not killed by the Each Choice test suite. One is not killed because the test cases' values where chosen in a way that the mutant does not make a difference in the output. By changing the input string to have six characters (so that the output is 3) this mutant can be killed. This mutant is not killed by the Base Choice test suite because it does not contain such a test case. The Pair Wise test suite however, contains such a test case and is able to kill this mutant. This is due to the selection of inputs for the test frame. If the values were more random to include inputs with higher numbers of characters, these mutants would be killed.

The second mutant is not killed by the Each Choice test suite because of the value that is used for `rlen`. This mutant changes `this.stringToReturn.length() / 2` to `this.stringToReturn.length()-2`. Since the Each Choice test suite only uses an `rlen` value of 4 (for all its test cases), which obviously has no effect on the output when executing this mutant, this mutant is not killed. By changing a test case to contain a greater value for `rlen`, this mutant will be killed. More variability in the selection of inputs values for test cases would have increased mutation score.

There is only one mutant that is not killed by the Pair Wise test suite. This mutant changes `(this.rlen - 1) / 2` to `(this.rlen - 1) - 2`. Since this test suite uses 5 for each occurrence of `rlen`, the two statements have equal behavior. Again, more variability in the selection of input values in a test suite would have increased the mutation score.

The Pair Wise test suite has a better overall mutation score (83%) and higher number of test cases (23) than the other two test suites (Table 13): 78.1% with 13 test cases for the Base Choice test suite and 67.1% with 11 test suites for the Each Choice test suite.

The mutation results are summarized in the following Table 9.

Criterion	Number of Test cases	Mutation Score	Mutation Score without Equivalent and untested functions
Base Choice	11	Live: 132 Killed: 447 Score: 77.0	Live mutants: 99 Killed mutants: 354 Mutation Score: 78.1
Each Choice	7	Live: 182 Killed: 397 Score: 68.0	Live mutants: 149 Killed mutants: 304 Mutation Score: 67.1
Pair Wise	13	Live: 110 Killed: 469 Score: 81.0	Live mutants: 77 Killed mutants: 376 Mutation Score: 83

Table 9 PackHexChar Case Study: Overall Statistics

As explained a large amount of test cases are not killed by the Each Choice test suite due to the limitation of the Each Choice criterion: it does not enforce specific choice combinations to be exercised, and therefore (because of a lack of chance) it does not. Three mutants are not killed because of deficiency in the selection of input. If a choice appears in several test frames and the choice specifies a range of values, one should try to select different values from the range for each occurrence of the choice in the test frame.

Table 10 shows the code coverage for each of the test suites. As can be seen from this table, the Each Choice test suite has the worst coverage percentage. The Pair Wise test suite has the highest code coverage.

Method	Statement			Branch			Loop			Term		
	BC	EC	PW	BC	EC	PW	BC	EC	PW	BC	EC	PW
PackHexChar	100	100	100									
convertStringToHex	100	100	100				66.7	66.7	66.7	58.3	100	100
equals	73.3	73.3	73.3	61.1	61.1	61.1	66.7	66.7	66.7	58.3	58.3	58.3
Hex2decimal	100	100	100				66.7	66.7	66.7	100	100	100
isHex	66.7	66.7	66.7	75	75	75	66.7	66.7	66.7	83.3	83.3	83.3
isHexStringChar	100	100	100									
oddDigitIsValidHex	100	100	100	100	100	100				100	100	100
packHexCharConvert	81.2	56.2	87.5	78.6	57.1	85.7				81.2	62.5	87.5
setInputString	0	0	0									
setOdd_digit	0	0	0									
setOutput	100	100	100				66.7	66.7	66.7	100	100	100
setRlen	0	0	0									
setValueToReturn	100	100	100									
valueToReturn	100	100	100	100	100	100				90	90	90
Total Coverage	**92.8**	**90.6**	**93.4**	**82.9**	**78.6**	**84.4**	**66.7**	**66.7**	**66.7**	**83.8**	**86.8**	**89.8**

Table 10 PackHexChar Case Study: Code Coverage

5.4 OrdSet Case Study

The following is a summary of the results obtained for the OrdSet case study. The result of the mutants is discussed in detail in Appendix E3. The count of mutants killed (or not) by the different test suites is illustrated by the Venn diagram of Figure 21.

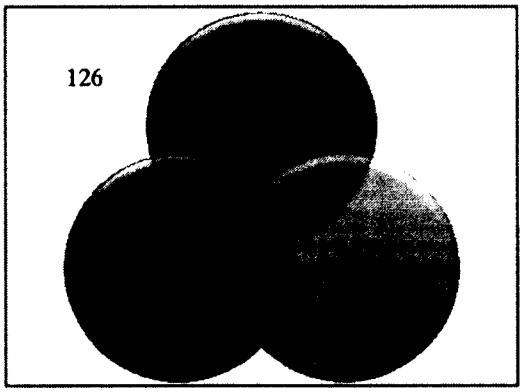

Figure 21: OrdSet Case Study: Venn diagram illustrating the counts of mutants killed by the different test suites

All three test suites give 100% statement and term coverage of the `OrdSetSimple` constructor. The loop coverage is 66.7% for all test suites (Table 12). This is because of the presense of some empty sets which results in some alternatives within the loop not being

exercised. There is a total of 26 mutants generated for the constructor. 20 are killed by all the test suites, 6 are not killed by any one of the test suites.

Since the `addElement` method is not tested specifically (instead, it is only tested indirectly when it is called by the `difference` method as we test this latter method), none of the test suites achieves 100% coverage for any of the three criteria. For instance, the code that checks the integer to add has a legal value (i.e., positive or null) is not exercised since all our test inputs have legal values. There is a total of 53 mutants generated for this method. 38 of the mutants are killed by all test suites while 15 remain alive. The mutants that remain alive are all seeded in the un-executed code.

All three test suites achieve 100% statement, branch, loop and term coverage for the `binSearch` method. 88 mutants are killed by all three test suites; 10 mutants are not killed by any of the test suites; nine of these mutants are equivalent while one is not. This mutant will be killed if the sets contained duplicate values, i.e., when one tries to add an integer value to a set that already contains this value. Since the CP specification is for the difference function and adding duplicate values to a set is not part of its specification, no specific test is created with such conditions, and this mutant is not killed. One mutant is killed by the Base Choice and Each Choice test suites but not by the Pair Wise test suite. The Base Choice and Each Choice test suites contain a test case where the difference set has only one value. Since the Pair Wise test suite does not contain such a test case the mutant is not killed by this test suite. One mutant is killed by the Base Choice and Pair Wise test suite but not by the Each Choice test suite. This mutant will be killed if an integer value located somewhere in the middle of the first set is in the second set. This is because the boundary for the binary search is changed and the item in the middle is missed. Since the focus of the case study is not to test the binary search function, there are no test cases in the Each Choice test suite specifically with such description.

Similarly to the `addElement` method, the `equals` method is not covered in its entirety by any of the test suites: error cases are not exercised. 55 mutants were generated for the `equals` method and all these mutants are alive. This method never gets executed.

For similar reasons, none of the test suites achieves 100% coverage (regardless of the criterion) of the `getElementAt` method. 30 mutants were generated for this method. 80% of the

mutants are killed by all test suites while 20% remain alive. The mutants that are alive are seeded in the un-executed code while killed mutants are in the executed code.

The `getSize` method is so simple that all three test suites achieve 100% coverage (regardless of the criterion) and kill all the ten mutants generated for this method.

The Base Choice and Pair Wise test suites achieve 100% coverage (for each criterion) of the `toString` method, while the Each Choice test suite achieves 66.7% loop coverage and 100% coverage for the other criteria. The Each Choice test suite executes a loop fewer times than the other two test suites. Nevertheless, we obtain the same mutation score for all test suites: 72.7% of the 22 generated mutants are killed by all test suites while 27.3% remain alive. The mutants that are alive are seeded in the un-executed code while killed mutants are in the executed code.

Since none of our test cases attempts to add an element to a set when this element is already in the set, one alternative of the implementation of the `make_a_free_slot` method is never executed. This is again due to the fact that we specifically test the `difference` method only. The Each Choice test suite gives a better coverage for `make_a_free_slot` method than the other two test suites since this test suite contains a test case where the values are not added to the set in increasing order of value. 35 mutants are killed only by the Each Choice test suite as a result. The next best test suite in terms of coverage is the Pair Wise test suite: better loop coverage than the Base Choice test suite; Statement, Branch and Term coverage are the same for both test suites. This does not mean that the loop in the code is not covered at all by the Base Choice test suite. It means that this part of the code gets executed fewer times than by the Pair Wise test suite.

76 of the 135 generated mutants are killed by all three test suites and 22 are not killed by any of the test suites. Out of the 22 mutants that are not killed by any test suite, 14 are equivalent while 8 are not killed because of one of the two following reasons: the part of the code where the mutant is seeded is not covered; the sets are not created with an attempt to add duplicate values in any of the test frames for the test suites, which is not part of the CP specification and there is no requirement for such test frame. The rest of the mutants are devided into three categories. First we have 35 mutants that are killed by the Each Choice test suite and not by the Base Choice and Pair Wise test suites. This is strictly due to the fact that the Each Choice test suite contains a

test case that creates one of the two sets used in the difference method by adding integers in an un-ordered way (e.g., if the set is to contain values 1, 4, 5, and 10, we tried to add them in the order 5, 1,10, 4). This selection of the input was not a result of using Category-Partition but was chosen as part of the experiment. And the mutants specifically killed by this test case are not seeded in the difference function, which we specifically test with the CP specification. Second, one mutant is killed by the Base Choice and Pair Wise test suites because these test sets contain an input set with consecutive values. This mutant causes the input of the `make_a_free_slot` function to be decreased by one. When adding values to a set, this function ensures that the value does not already exist in the set. If we have two consecutive items in the set (i.e. 1, 2), the mutated program adds the first item to the set, then the second value is decreased by one (making it equal to the previous value) and this item is not added to the set since it already exists. The Base Choice and Pair Wise test suites contain a set with consecutive values because of the way the inputs where chosen. The Each Choice test suite can kill this mutant as well if it had the same input. If the values of inputs are chosen such that we exercise consecutive as well as non-consecutive values, this mutant will be killed. Third we have one mutant that is only killed by the Pair Wise test suite. This mutant resides in the `make_a_free_slot method`. When adding values to a set, this function makes sure the set does not already contain the value and finds a place for the new value so that the set is ordered. This particular part of the code identifies the index in the ordered set where to place the new element, i.e., the index of the set where all the elements with a lower index have a smaller value than this new element. This part of the code goes through a loop and compares the integer value to be added, referred to as newValue, to the integer values in the set at increasing indexes, referred to as existValue(i). Each time the loop iterates the mutant decreases newValue by one after each comparison: i.e., if we have 4 elements in the set, newValue is decreased 3 times by 1 before the 4^{th} comparison and the last comparison is therefore with newValue-3. This results in unordered sets under a specific condition which we identified to be the following: assuming we have to values V1 and V2, V2>V1 to be added in the set, and V1 being at index i in the set, then if V2-1>V1 the mutant produces an unordered set (value V2 will be inserted before value V1). Having unordered sets in memory is not enough to reveal the fault. The difference function looks at all the elements of the first set in sequence and tries to find them in the second set with a binary search. Regardless of whether the first set is ordered or not in memory, this does not help revealing the fault. We have identified we need the

second set to be unordered, i.e., to satisfy the condition above. This is however not sufficient. We also need the common part of the two sets to be in the unordered part of the second set. This is even not sufficient. In fact, in order to reveal the fault (i.e., kill the mutant), we need the binary search to fail because the second set is unordered in memory. Since characterizing this condition would take too much time and space in this thesis, we do not further discuss the condition under which the mutant would be killed. It is sufficient to say that, at this stage of the discussion, it is difficult to imagine a CP specification that would necessarily lead to exercising this condition. This therefore appears to be simply an input selection issue.

All the test suites achieve 100% coverage of the `difference` method for the Statement, Branch and Term criteria. 25 mutants were generated for the difference method. 96% of the mutants are killed by all test suites while 4% (one) remain alive. The mutant that remains alive is equivalent. The CP specification being specifically focusing on the difference function, we find no difference between the selection criteria in terms of effectiveness.

The overall mutation score is summarized in Table 11. The code coverage percentages are given in Table 12.

Criterion	Number of Test cases	Mutation Score	No Equivalent and non-used methods (equals method) Mutation Score
Base Choice	7	Live: 162 Killed: 298 Score: 64.8	Live: 93 Killed: 298 Score: 76.2
Each Choice	5	Live: 128 Killed: 332 Score: 72.2	Live: 59 Killed: 332 Score: 84.9
Pair Wise	8	Live: 163 Killed : 297 Score: 64.5	Live: 94 Killed : 297 Score: 76.0

Table 11 OrdSet Case Study: Overall Statistics

Method	Statement			Branch			Loop			Term		
	BC	EC	PW	BC	EC	PW	BC	EC	PW	BC	EC	PW
OrdSetSimple	100	100	100	-	-	-	66.7	66.7	66.7	100	100	100
addElement	37.5	37.5	37.5	50.0	50.0	50.0	-	-	-	50	50	50
binSearch	100	100	100	100	100	100	100	100	100	100	100	100
difference	100	100	100	100	100	100	33.3	66.7	66.7	100	100	100
Equals	0	0	0	0	0	0	0	0	0	0	0	0
getElementAt	50	50	50	50	50	50	-	-	-	50	50	50
getSize	100	100	100	-	-	-	-	-	-	-	-	-

make_a_free_slot	62.5	87.5	62.5	66.7	83.3	66.7	66.7	100	100	66.7	91.7	100
toString	100	100	100	-	-	-	100	66.7	100	100	100	100

Table 12 OrdSet Case Study: Code Coverage

5.5 TaxiBilling Case Study

There are 316 mutants generated for classes `TaxiBilling` and `TaxiDaoStructure`. 123 mutants are killed by all three test suites and 81 are not killed by any of the test suites. 30 of the live mutants are equivalent. The count of mutants killed (or not) by the different test suites is illustrated by the Venn diagram of Figure 22. The details of why each mutant is killed or is alive are in Appendix F3.

The Pair Wise test suite has a better overall mutation score (79%) and higher number of test cases (23) than the other two test suites (Table 13): 53.5% with 13 test cases for the Base Choice test suite and 56.3% with 11 test suites for the Each Choice test suite.

We next discuss separately the two classes of the system.

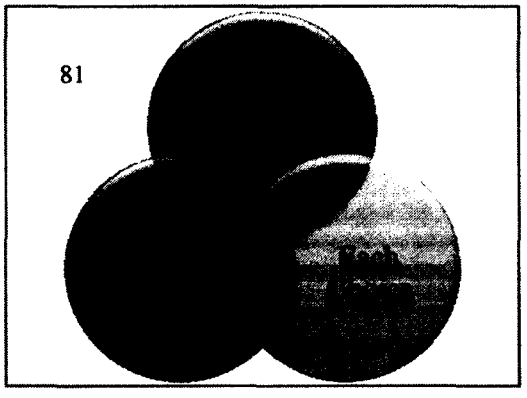

Figure 22 TaxiBilling Case Study: Venn diagram illustrating the counts of mutants killed by the different test suites

5.5.1 TaxiBilling:

The Base Choice test suite has a higher mutation score for class TaxiBilling than the other two test suites (Table 13). This is explained by a difference in structural (i.e., statement, branch and term) coverage: the Base Choice test suite achieves 100% statement coverage while the Each Choice and Pair Wise test suites only achieve 88.9% statement and 87.5% branch and term

coverage. The details of the coverage for the TaxiBilling class and TaxiDaoStructure class can be found in Table 14 and Table 15, respectively.

19 mutants seeded in this class are not killed by any of the test suites. 15 of these mutants are equivalent while four are not killed because of the way the constructors of the class are exercised by our test cases. Since the target of the test cases is not to test the constructors specifically, but rather to test the main functionality, change in inputs will not make a difference in whether these mutants are killed or not.

30 mutants are killed by all three test suites. Two mutants are not killed by the Each Choice test suite because the Each Choice test suite does not contain a test frame where the value of `holiday` is true. This is a consequence of the test frames built by the Each Choice criterion and changing the input will not have an impact in killing the mutant. But a different set of test frames, adequate for the Each Choice criterion could be constructed and could result in killing these mutants.

One of the three mutants only killed by the Base Choice test suite is in the `TaxiBilling` class. This mutant makes changes to the part of the program that involves the minimum cost for a trip. Since there is a test case with the value of totalCost less than the specified minimum this mutant is killed. The Each Choice and Pair Wise test suites could also kill this mutant if they had a test case satisfying such a condition. Having a test case with input values leading to a totalCost of less than the minimum would allow the Each Choice and Pair Wise test suites to achieve 100% code coveragefor the `Calculate` method.

To conclude the discussion about class TaxiBilling, the higher structural coverage and better mutation score of the Base Choice test suite is due to the selection of test inputs from test frames. Similar test inputs could have been chosen for the Each Choice and Pair Wise sets of test frames and this would have increased the coverage and mutation scores to the levels of the one achieved by the Base Choice test suite.

5.5.2 TaxiDaoStructure:

69 mutants are not killed by any of the test suites. 19 of these mutants are equivalent. The remaining ones fall in one of the following categories.

Five mutants require the value of `departureTime` to be an invalid value leading to an error choice. These invalid values include an error value on the boundary of being an error, and an error value that is not at the boundary. An example is if we have a valid range [0,23], 24 would be an error choice at the boundary and 25 is an error choice not at the boundary. Since there is only one test frame built for each error choice, only one of the mutants will be killed by each of the test suites. Killing these mutants will require more test frames with error choices. If there are more error choices in the CP specification building these test frames will be possible.

Four mutants require the value of `departureTime` to be between 0 and 7 (inclusive) and the day to be `Sunday`. The Each Choice test suite covers each choice once and it does not require such combination and in fact it does not exercise it and therefore missed these mutants. The Pair Wise test suite however exercises this combination in one test frame since it covers pairs of choices and therefore kills these mutants. The Base Choice test suite does not contain a test frame with day equal to `Sunday` because `Sunday` cannot be combined with the base choice of category Cat2 (choice Ch2.4) in test frames due to constraints. Therefore the Base Choice test suite is unable to kill these mutants.

Three mutants require holiday to be true. One of these additionally requires the value of `departureTime` to be 23 (which is a boundary value). All test suites can kill this mutant in theory. However, making the Each Choice and Base Choice test suites kill this mutant would mean changing the value of `departureTime` to 23 in the only test frame that involves the holiday choice, and this would result in the test suites missing mutants that are currently killed. The Pair Wise test suite is the only test suite with more than one test frame involving the holiday choice therefore there are fewer mutants that would be affected by this change of `departureTime` value. Killing two other mutants requires combining the holiday choice, the weekday choice, a `departureTime` choice corresponding to ranges [17, 19] or [19, 24], and the `destArea` choice different from urban and suburban. Since this combination of choices is not enforced by any of the criteria, no criterion ensures this combination happens and there are little chances such a situation would happen when selecting test inputs. The question is then whether the CP specification should have be such that this combination is enforced (by at least one criterion).

The rest of the mutants not killed by any of the test suites require the day to be a `WeekDay` and not a `holiday`. Two of these mutants are killed if we have boundary values for `departureTime` (i.e., 23 and 7). All test suites can kill these mutants with change in `departureTime`. Other mutants in this category require combinations of three or four choices. None of the criteria ensures such combinations would be exercised. Therefore if the test inputs are changed to accommodate these mutants, another combination might be missed resulting in other mutants not killed.

46 mutants are killed by the Pair Wise test suite but not by the Each Choice and Base Choice test suites. Since the Pair Wise test suite covers more combinations of choices than the other two, it is able to kill more mutants. Only eight mutants are not killed by the Pair Wise test suite but are killed by the Each Choice or Base Choice test suites (or both). Four mutants are only killed by the Each Choice test suite, two mutants are only killed by the Base Choice test suite and two are killed by both the Base Choice and Each Choice test suites. The mutants killed only by the Each Choice test suite require the value of `departureTime` to be at boundary value 7, the value of holiday to be false and the area to be urban: this would result in different behaviours between the mutant and the original program with respect to the applicable charge return value. It is by chance that the Each Choice test suite contains such a test case. This requires three choices to be covered at the same time and no test suite has this requirement. The two mutants only killed by the Base Choice test suite require a weekday that is not a holiday, a destination that is urban and a departureTime at boundary value of 0. This mutant is killed because the combination is enforced in the Base Choice test suite (since all these are base choices except `departureTime`) and the boundary value is chosen during input selection. This combination however is not enforced by the Pair Wise test suite and therefore this is not killed. The two other mutants killed by the Base Choice and the Each Choice test suites require weekDay and not a holiday and the departureTime value between 0 and 7. The Pair Wise test suite contains a test case only at the boundary (7) with the mentioned description but not a value in the range. Therefore adding a value at the boundary which is 7 will enforce this combination and this test suite can kill these mutants.

32 mutants are not killed by the Base Choice test suite but are killed by the Each Choice and Pair Wise test suites. 25 of them require the day value to be `Sunday` and since the Base Choice

test suite does not contain a test frame with `Sunday` (as explained above), these mutants cannot be killed. Any given test frame in the Base Choice test suite only has one choice that is not a base choice. The 7 remaining mutants not killed by Base Choice require more than one choice to be non-base. Therefore these mutants cannot be killed by this test suite.

25 Mutants are killed by the Base Choice and Pair Wise test suites but not by the Each Choice test suite. Some test frames could lead to selecting test input that would kill these mutants. However, only one test input selection is done for each test frame and changing the values of inputs will cause other mutants, which are currently killed, to remain alive. Since the Each Choice has a limited number of test frames, not every day of the week is covered (the implementation checks for every single day of the week: Monday, Tuesday...; which is unexpected). Since all these values are in a choice weekDay, there is no requirement for this coverage. However mutants require all days of the week to be exercised. This limitation also causes some mutants to stay alive. Some mutants will be killed if the value is chosen at a boundary of a specific range (i.e. departureTime >=0). If the CP specification defined 0 as a boundary, in a specific choice, these mutants would be killed. The requirement for the Each Choice test suite is to have each choice present in at least one test frame. This means that the Each Choice test suite will only be able to have one value in a given range for a choice in a given test frame. This is a result of the CP specification not containing all boundary values in specific choices.

The Pair Wise test suite has a higher mutation score than the Base Choice and Each Choice test suites. This is because specific combinations of inputs need to be exercised in order to kill many mutants. Since the Pair Wise test suite is the only test suite that systematically covers pairs, the test suite exercises more choice combinations than the other test suites, and there are more mutants killed by this test suite. Eight mutants will be killed if combinations of three choices were covered by a test suite.

The Each Choice test suite gives a better coverage than the Base Choice test suite. As mentioned earlier a large amount of mutants not killed by the Base Choice test suite is due to the lack of presence of a crucial value in the suite (i.e. `Sunday`). Since a test frame in the Base Choice test suite only contains one choice that is non-base, the combination of the non-base value and all the other base choices may be infeasible. This is the case in our case study: `Sunday`

that cannot be combined with Ch2.4, which is a base choice. This leads us to argue that tool support for criteria specific to Category Partition should report on unfeasible objectives (in this case combining all base choices except one, with one non-based choice) to the user in addition to providing test frames. The user may then decide to add a test frame to exercise the choices that would not be exercised otherwise. The Each Choice test suite however does not have this limitation and it is required to cover every choice at least once. Therefore it performs better in this aspect. The nature of the Each Choice test suite also allows it to have test frames that have more random combinations than the Base Choice test suite where combinations of choices are known ahead of time.

Criteria	Frames	TaxiBilling Mutations	TaxiDaoStructure Mutations	Combined result	No Equivalent Mutants
Base Choice	13	Live: 19 Killed:33 Score: 63.5	Live: 144 Killed: 120 Score: 45.5	Live: 163 Killed: 153 Score: 48.4	Live: 133 Killed: 153 Score: 53.5
Each Choice	11	Live: 23 Killed: 29 Score: 55	Live: 132 Killed: 132 Score: 50	Live: 155 Killed: 161 Score: 51	Live: 125 Killed: 161 Score: 56.3
Pair Wise	23	Live: 20 Killed: 32 Score: 61.5	Live: 70 Killed: 194 Score: 73.5	Live: 90 Killed: 226 Score: 71.5	Live: 133 Killed: 226 Score: 79

Table 13 TaxiBilling Case Study: Overall Statistics

Method	Statement			Branch			Loop			Term		
	BC	EC	PW	BC	EC	PW	BC	EC	PW	BC	EC	PW
setTaxiDao	100	100	100	-	-	-	-	-	-	-	-	-
parseWeekDay	100	100	100	100	100	100	-	-	-	-	-	-
Calculate	100	100	88.9	100	87.5	87.5	-	-	-	100	87.5	87.5
TaxiBilling	100	43.6	100	-	-	-	-	-	-	-	-	-
TaxiBilling	100	100	100	-	-	-	-	-	-	-	-	-

Table 14 TaxiBilling Case Study: Code Coverage of TaxiBilling Class

Method	Statement			Branch			Loop			Term		
	BC	EC	PW	BC	EC	PW	BC	EC	PW	BC	EC	PW
populateInMemoryValues	100	100	100	-	-	-	-	-	-	-	-	-
findMinimumCharge	100	100	100	-	-	-	-	-	-	-	-	-
findInitialCharge	100	100	100	-	-	-	-	-	-	-	-	-
findApplicableCharge	43.6	43.6	74.4	44.3	50	71.6	-	-	-	48.2	51.8	70.2
TaxiDaoStructuralImpl	100	100	100	-	-	-	-	-	-	-	-	-

Table 15 TaxiBilling Case Study: Code Coverage of TaxiDaoStructuralImp Class

5.6 Discussion of Results

As can be seen from the case studies's results, there are a few factors that cause mutants not to be killed by a test suite. These include:

1. The mutant is equivalent and does not have any impact on the functionality and therefore cannot be discovered.
2. The CP specification does not reflect the system's behavior entirely, for instance not all the boundaries of ranges belong to separate choices.
3. The selection of inputs for a given test frame are not varied enough and do not lead to discovering some mutants.
4. The test frames built by the criterion do not allow the selection of inputs that would ensure some mutants are killed.

We have taken the equivalent mutants out of the results because these cannot be killed by any test suite and pay no value to the result analysis. There are several cases where the Category Partition specification does not reflect the system's behavior. The test suites constructed for the triangle case study would have led to better results if there where choices considering the comparison of two inputs as well as the three in triangle inequalities (i.e., SA<SB). For the NextDate case study adding choices with boundary values to the CP specification would cause more mutants to be killed. This includes the value 28 for the day. If the CP specification for the PackHexChar case study included a choice with the value of inputString containing characters A-F, more mutants would be killed. The test suites for TaxiBilling case study would have killed more mutants if there were choices at boundary values as well. We have observed that for the TaxiBilling system we would kill more mutants if the boundaries were chosen differently, specifically resulting in more than one error choice. This is observed for the case where an error choice is a value for hour that is greater or equal to 24. The range specified for an error range in the current CP specification is one that is greater than or equal to 24. If this choice were split into two choices, one for 24 and one for values greater than 24, these mutants would be killed. This is a problem with defining boundary values for choices in the CP specification.

The selection of inputs for a given test frame for each of the test suites also plays a major role in the results. We observe that choosing inputs, not only at the boundaries of their own

range, but at the boundaries of the domains they define with respect to outputs, for example an input that is isosceles but very close to an equilateral triangle, would kill more mutants. For the PackHexChar case study, more mutants would be killed if the inputs where chosen more randomly to include outputs of greater length. The Each Choice test suite for the OrdSet Case study included a choice that had an unordered set and caused many mutants to be killed that are not killed by the other two test suites. Selecting different values from the range for each occurrence of the choice in the test frame results in more mutants killed.

As observed from the case studies, some of the selection criteria do not allow the inputs to kill mutants. An example is the TaxiBilling case study where the mutants could not be killed by the Base Choice test suite because it did not have a value of `Sunday` as an input.

The results of the case studies allow us to observe the following behaviour (Table 16):

1. The Pair Wise Criterion gives better results in all case studies except for the OrdSet case study where the Each Choice criterion performs better. This exception is due to the selection of inputs which is better for the Each Choice criterion than for the Pair Wise criterion in this case study.

2. The Base Choice criterion gives better results than the Each Choice criterion for Triangle and PackHexChar due to the nature of the test frames. The test frames in the Base Choice test suite for these particular case studies cover more output values (or output parameter equivalence classes) than the Each Choice test suite. If another Each Choice test suite were constructed, with a different test input selection strategy, for these case studies it may have given better results.

3. The Each Choice criterion gives better results than the Base Choice criterion for NextDate, OrdSet and TaxiBilling. This is due to the nature of selection of choices for a frame. Specifically, the Each Choice test suite allows more random choices to be chosen for a test frame resulting in more mutants being killed.

4. The Pair Wise test suite performs better with a reasonably higher cost, i.e., it does not require many more test cases. One exception is the OrdSet case study where the Each Choice test suite performs better due to the selection of inputs. With better selection of

inputs, the Pair Wise test suite would have killed more mutants than the Each Choice test suite.

	Triangle		NextDate		PacHexChar		OrdSet		TaxiBilling	
	Score	Cost	Score	Cost	Score	Cost	Score	Cost	Score	Cost
Base Choice	81.0	9	56.5	9	78.1	11	76.2	7	53.5	13
Each Choice	66.0	7	62.6	7	67.1	7	84.9	5	56.3	11
Pair Wise	85.0	10	81.8	10	83	13	76.0	8	79	23

Table 16 Comparison of mutation scores and costs (i.e., number of test cases) for all case studies

5.7 Threats to Validity

As usual in any experiment, results have to be considered with care as there are possible threats to validity.

Threats to *conclusion validity* are concerned with issues that affect the ability to draw the correct conclusion. In our case, they could be essentially due to small number of data points, which prevented us from performing statistical analyses. For this reason we did not rely too much on statistical results and complemented the quantitative analysis with a detailed qualitative analysis.

An *internal validity* threat exists when the outcome of the experiment may not necessarily be caused by the treatment applied, but can be caused by another factor not controlled in the experiment. We were careful in conducting the testing activities similarly for each case study system. Deviations especially in the way we selected inputs were reported and their impact on results discussed. Obviously, other reasons could explain our observations. More case studies would be necessary to limit further this threat.

External validity relates to the external aspects that interact with the treatments and limit the generalization of the results. This threat is mainly related to our selection of case study systems, the CP specifications and the test input selection procedures. Although we were careful, more case studies are necessary to further limit this threat.

Construct validity is mainly related to our use of mutants to measure fault detection effectiveness. However, existing literature (e.g., [36, 37]) suggests that faults seeded using mutation operators can be representative of real faults, and relying on mutation to compare test techniques is practical. The fact that we used five case study systems with very different code characteristics—thus leading to very different samples of mutants—should, however, limit the likelihood of this threat.

6 Conclusion

Melba has been designed as a tool to help users understand a test suite, identify the needs for improvement and applying modifications. This tool uses the Category Partition black box testing method, along with a machine learning algorithm. However Melba has not been originally designed to construct test suites.

This thesis adds a new feature to Melba which gives the users the option to construct test frames using any of the Base Choice, Each Choice and Pair Wise selection criteria that are specific to Category Partition. Test frames can be considered test cases specifications since they specify conditions on test inputs. These are two of the main contributions of the thesis where 1) The user can construct test frames by defining a CP specification 2) Selection criteria are taken into consideration when constructing the test suites.

Taking constraints into consideration, we would like to have an effective and fast algorithm for obtaining test suites with the coverage criterion. There has been research on construction of test suites but none of them fits our work better than Covering Arrays by Simulated Annealing (CASA). As the name suggests this tool uses a simulated annealing (optimization) algorithm to perform Combinatorial Interaction Testing (CIT), taking constraints into consideration. We have chosen this tool because it is effective, fast, takes constraints into consideration and has readily available code to use. CASA is used for t-way testing meaning that it can only be used for obtaining Each Choice (1-way) and Pair Wise (2-way) test suites. We have made changes to CASA to be able to construct test suites for the Base Choice criterion as well.

There has been little work done on the comparison of the three selection criteria in the presence of constraints. Our next major contribution is therefore an experimentation using the test frames built using Melba and CASA for the three selection criteria and applying them to five case studies.

Using the CP specification, we have used Melba to construct test suites for the five case studies using each of the three criteria. After the construction of test suites, test cases were built by using a specific input value selection strategy, i.e., a specific strategy to identify input values that satisfy the conditions expressed by the choices making test frames. Mutants were then

generated using MuJava and the test cases were run against these mutants to measure effectiveness. When compairing test suites and therefore criteria, we also measured code coverage, using the CodeCover tool, which gives statement, branch, loop and term coverage for the code.

We observed that the failure to discover mutants can be a result of one of the following:

1. The mutant is equivalent and does not have any impact on the functionality and therefore cannot be discovered.
2. The CP specification does not reflect the system's behavior entirely.
3. The selection of inputs for a given test frame are not very random and do not lead to discovery of mutants.
4. The test frames built by the criterion do not allow the inputs for the mutants to be killed.

After the analysis of mutation scores we observe that the Pair Wise test suite performs better in all cases but one case study where the Each Choice test suite performs better than all test suites. Pair Wise seem to be the most cost effective criteria: highest fault detection capability, cost reasonably higher than the other two criteria. We have concluded that with the presence of constraints, the Base Choice test suite will not always perform better than the Each Choice test suite.

6.1 Future Work

For the design of Melba future work can include having a better solution for the integration of Melba and CASA. Currently the two tools interact through files, which is far from being ideal. We however only wanted a proof of concept in this thesis.

We have used manual means for constructing test cases i.e., for selecting input values that satisfy the conditions specified in test frames. For future work a more automated way to construct test cases would be desirable. SAT solvers can be considered to (partially) solve this issue.

We have compared three selection criteria. Future work would be to compare more criteria such as 3-way coverage, which would allow the tester to have more options in choosing a criterion that better fits their needs.

Since we also observed that the Category Partition specification, the construction of an adequate set of test frames for a criterion, and the selection of test inputs all have impacts on the cost-effectiveness of the Category Partition testing technique, more experiements are warranted to try alternative Category Partition specifications, alternative mechanisms to construct an adequate set of test frames for a criterion, and alternative test inputs selection mechanisms.

One other observation we made is that it is important to ensure, either with the CP specification, the test frame construction mechanism, or the test input selection (or a subset of those) that different output values for the parameters, and actually different output equivalence classes for the output parameters, be exercised by adequate test suites. To address this issue, we can foresee to possible routes. First, we can try to incorporate output equivalence classes to the Covering Array specification. However, we would also need to ensure that combinations of columns in the array are feasible: i.e., the combination of choices for inputs does reasult in the output equivalence class (another column on the array). Second, we can try to rely on Melba's test suite re-engineering capability. However, the larger the test suite the better Melba performs since it calls a machine learning algorithm. Only further studies will tell what is feasible.

Obviously, more complex, industrial sized case studies should also be used to compare criteria in experiments. Fortunately, the Melba+CASA tool now provides a good starting point infrastructure to conduct such experiments.

References:

[1] Ghezzi C., Jazayeri M., and Mandrioli D., Fundamentals of Software Engineering, Englewood Cliffs: Prentice Hall, 1991.

[2] Ammann P., Offutt J., Introduction to Software Testing, Cambridge: Cambridge University Press, 2008.

[3] Balcer M., Hasling W., Ostrand, T., "Automatic Generation of Test Scripts from Formal Test Specifications," in *Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, 1989.

[4] Amman P. ,Offutt Jeff, "Using Formal Methods to Derive Test Frames in Category-Partition Testing," in *In Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS, Gaithersburg, MD)*, Gaithersburg, 1994.

[5] Chan Eric Ying Kwong, Poon, Pak-Lok, Yu, Yuen Tak, "On the Testing of Particular Input Conditions," Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004

[6] Chen T.Y., Poon P.L.,Tse T.H., "A Choice Relation Framework for Supporting Category-Partition Test Case Generation," in *IEEE Transactions on Software Engineering*, 2003.

[7] Chen T.Y., Poon P.L. ,Tang S.F. , and T.H., "On the Identification of Categories and Choices for Specification-Based Test Case Generation," in *Information and Software Technology*, 2004.

[8] Grochtmann M. and Grimm K., "Classification Trees for Partition Testing," STVR, 3 (2), pp. 63-82, 1993.

[9] Grochtmann M., Grimm K. and Wegener J., "Tool-supported test case design for black-box testing by means of the classification- tree editor," Proc. EuroStar, pp. 169–176, 1993.

[10] Lehmann E. and Wegener J., "Test case design by means of the CTE XL," Proc. EuroStar, pp. 1-10, 2000.

[11] Poon P.-L., Chen T. Y. and Tse T. H., "Choices, Choices: Comparing between CHOC'LATE and the Classification-Tree Methodology," Proc. Ada-Europe, Lecture Notes in Computer Science, pp. 162-176, 2012.

[12] N. R. Traldi Spido, "Method and tool support for refinement of test suites," Carleton University, Ottawa, 2010.

[13] Bacle T.J., Ostrand and M.J., "The Category-Partition Method for Specifying and Generating Functional Test," *Communications of the ACM,* vol. 31, no. 6, pp. 676-686, 1988.

[14] J. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

[15] Witten I. H. and Frank E., Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufman, 2005.

[16] Z. Bawar, "Using Machine Mearning to Refine Black-Box Test Specifications and Test Suites," Carleton University, Ottawa, Canada, 2009.

[17] Briand, L.C., Labiche, Y., Bawar, Z., "Using Machine Learning to Refine Black-Box Test Specifications and Test Suites," in *International Conference on Quality Software*, 2008.

[18] Briand, L.C., Labiche, Y., Bawar, Z., Spido, Nadia, "Using machine learning to refine Category-Partition test specifications and test suites," *Information and Software Technology,* vol. 51(11), pp. 1551-1564, 2009.

[19] Mats Grindal, Jeff Offutt and Sten F. Andler, "Combination Testing Strategies: A Survey," *Software Testing Verification and Reliability,* vol. 15, p. 167–199, 2005.

[20] Danziger, L. Moura, E. Mendelsohn, B. Stevens , "Covering Arrays Avoiding Forbidden Edges," *Theoretical Computer Science 410,* pp. 5403-5414, 2009.

[21] Garvin B.J., Cohen M. B.and Dwyer M. B., "Covering Arrays by Simulated Annealing (CASA)," University of Nebraska-Loncoln, [Online]. Available: http://cse.unl.edu/~citportal/tools/casa/source2.php [August 20, 2012].

[22] C. C. Coello, "Theoretical and Numerical Constraint Handling Techniques Used with Evolutionary Algorithms: A Survey of the State of the Art," *Competer Methods in Applied Mechanics and Engineering,* vol. 11, no. 191, pp. 1245-1287, 2002.

[23] K. R. ,. K. D. R. O. V. a. L. J. Lei Y., "IPOG: A General Strategy for t-way Software Testing," in *Engineering of Computer-Based Systems, IEEE International Converence on the,*, 549-556, 2007.

[24] Cohen M. B., Gibbons P. B. , Mugridge W. B. , and Colbourn C.J., "Constructing Test Suites for Interaction Testing," in *Proceedings. 25th International Confrence on*, 2003.

[25] J. Stardom, "Metaheuristics and the Search for Covering and Packing Arrays.," *Master's thesis, Simon Fraser University,* 2001.

[26] Garvin Brady J., Cohen Myra B. , Dwyer Matthew B., "An Improved Meta-Heuristic Search for Constrained Interaction Testing," in *International Symposium on Search-Based Software Engineering (SSBSE)*, 2009.

[27] P. C. Jorgensen, in *Software testing [electronic resource] : a craftsman's approach, third edition*, Norwood, Mass, Auerbach Publications, 2008.

[28] Briand L.C., Di Penta M., Labiche Y., "Assessing and Improving State-Based Class Testing: A Series of Experiments," in *IEEE Transactions of Software Engineering 30(11)*, 2004.

[29] Ma Yu-Seung, Offutt Jeff and Kwon Yong-Rae, "MuJava : An Automated Class Mutation System," *Journal of Software Testing, Verification and Reliability,* vol. 15(2), pp. 97-133, June 2005.

[30] Ma Yu-Seung, Offutt Jeff, "Description of Class Mutation Mutation Operators for Java," Fairfax, 2005.

[31] Ma Yu-Seung, Offutt Jeff, "Description of Method-level Mutation Operators for Java," Fairfax, 2005.

[32] R. Schmidberger, "CodeCover," [Online]. Available: http://www.codecover.org/. [August 20, 2012].

[33] Hayhurst, Kelly; Veerhusen, Dan; Chilenski, John; Rierson, Leanna , "A Practical Tutorial on Modified Condition/ Decision Coverage," *NASA,* May 2001.

[34] Abramovici, Miron; Breuer, Melvin A; Friedman, Arthur D, Digital Systems Testing and Testable Desing, Computer Scienve Press, 1990.

[35] Ostrand T.J. and Bacler M.J., "The Category-Partition Method for Specifying and Generating Functional Test," *Communications of the ACM,* vol. 31, no. 6, pp. 676-686, 1988.

[36] Andrews J. H., Briand L. C. and Labiche Y., "Is Mutation an Appropriate Tool for Testing Experiments?," Proc. IEEE ICSE, pp. 402-411, 2005.

[37] Andrews J. H., Briand L. C., Labiche Y. and Namin A. S., "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," IEEE TSE, 32 (8), pp. 608-624, 2006.

APPENDIX A USE CASE DESCRIPTIONS FOR MELBA

Use Case Name	**UC1 Load Initial Data**	
Brief Description	This use case is used to load initial data that is CP specifications (input and outputs) and also a test suite file defined by the user.	
Precondition		
Primary Actor	User	
Secondary Actor	None	
Dependency	INCLUDE USECASE UC1.1 Load Test Suite INCLUDE USECASE UC1.2 Define CP Spec – Input INCLUDE USECASE UC1.3 Define CP Spec – Output	
Generalization		
Basic Flow	**Steps**	
	1	The user creates a new project by providing a name for the project
	2	The user loads the test suite
	3	The user defines system CP input specs
	4	The user defines system CP output specs
	Postcondition	The Initial data has been loaded.
Specific Alternative Flows	None	
Global Alternative Flows	None	
Bounded Alternative Flows	None	

Table 17 UC1 Load Initial Data

Use Case Name	**UC1.1 Load Test Suite**	
Brief Description	This test case is used to load test suite using a TTCN3 file provided by the user	
Precondition	User has created a new project	
Primary Actor	User	
Secondary Actor	None	
Dependency	EXTENDED BY USE CASE UC1.4 Define Parameter EXTENDED BY USE CASE UC1.5 Verify Parameter Compatibility	
Generalization	None	
Basic Flow	**Steps**	
	1	The user provides the test suite in form of a TS file
	2	The system identifies parameters and types
	3	The system loads the parameters and types into CP Specification
	4	The system VALIDATES that parameters and types entered by the user match the TS file provided
	Postcondition	The test suite is successfully loaded.
	BFS2	
	1	The user provides the test suite in form of a file that is not TS
	2	The system is unable to open file
	Postcondition	The system prompts an error message to the user to enter a correct file.
	Steps	
	1	The user defines parameters and types by entering these values
	2	The system loads parameters and types

		from the file provided by the user
	3	The system VALIDATES that parameters and types entered by the user match the TS file provided
	Postcondition	The test suite is successfully loaded
Bounded Alternative Flows		

Table 18 UC1.1 Load Test Suite

Use Case Name	**UC1.2 Define CP Spec – Input**	
Brief Description	In this use case the user defines the test input parameters and environment variables.	
Precondition	None	
Primary Actor	User	
Secondary Actor	None	
Dependency	INCLUDE USE CASE UC1.2.1 Define Category and Choices INCLUDE USE CASE UC1.2.2 Select Base Choice EXTENDED BY USE CASE UC1.4 Define Parameter EXTENDED BY USE CASE UC1.5 Verify Parameter Compatibility	
Generalization	None	
Basic Flow	**Steps**	
	1	INCLUDE USE CASE Define Category and Choices
	2	INCLUDE USE CASE Select Base Choice
	3	IF the test suite file is loaded Then The user defines categories and choices, and base choice for each of the categories.
	4	The parameter is added to the system
	Postcondition	The CP Specs are added to the system
Specific Alternative Flows	Steps	
	1	EXTENDED BY USE CASE Define Parameter
	2	IF the test suite file is not loaded yet, THEN the user defines parameters, Categories and choices, and base choice for each of the categories ENDIF
	Postcondition	The CP Specs are added to the system
Global Alternative Flows	None	
Bounded Alternative Flows	None	

Table 19 UC1.2 Define CP Spec – Input

Use Case Name	**UC1.2 Define CP Spec – Input**	
Brief Description	In this use case the user defines the test input parameters and environment variables.	
Precondition	None	
Primary Actor	User	
Secondary Actor	None	
Dependency	INCLUDE USE CASE UC1.2.1 Define Category and Choices INCLUDE USE CASE UC1.2.2 Select Base Choice EXTENDED BY USE CASE UC1.4 Define Parameter EXTENDED BY USE CASE UC1.5 Verify Parameter Compatibility	
Generalization	None	
Basic Flow	**Steps**	
	1	INCLUDE USE CASE Define Category and Choices
	2	INCLUDE USE CASE Select Base Choice
	3	IF the test suite file is loaded Then The user defines categories and choices, and base choice for each of the categories.
	4	The parameter is added to the system
	Postcondition	The CP Specs are added to the system
Specific Alternative Flows	Steps	
	1	EXTENDED BY USE CASE Define Parameter
	2	IF the test suite file is not loaded yet, THEN the user defines parameters, Categories and choices, and base choice for each of the categories ENDIF
	Postcondition	The CP Specs are added to the system
Global Alternative Flows	None	
Bounded Alternative Flows	None	

Table 20 UC1.2 Define CP Spec – Input

Use Case Name	**UC1.2.1 Define Category and Choices**	
Brief Description	This use case is for the user to define categories and choices for each parameter.	
Precondition	The user has defined at least one parameter	
Primary Actor	User	
Secondary Actor	None	
Dependency	EXTENDED BY USE CASE UC1.6 Select Pre-defined Operation EXTENDED BY USE CASE UC1.7 Write JAVA Expression	
Generalization	None	
Basic Flow	**Step**	
	1	The user selects a parameter
	2	The user enters a name for a category

	3	The user enters a name for a choice
	Postcondition	The CP input specifications are defined
Specific Alternative Flows	Step	
	1	EXTENDED BY USE CASE Select Pre-defined Operation
	2	EXTENDED BY USE CASE Write JAVA Expression
Global Alternative Flows	None	
Bounded Alternative Flows	None	

Table 21 UC1.2.1 Define Category and Choices

Use Case Name	**UC1.2.2 Select Base Choice**	
Brief Description	This use case is for the user to select a base choice.	
Precondition	The user selects to define CP specifications The user has defined at least one choice	
Primary Actor	User	
Secondary Actor	None	
Dependency	None	
Generalization	None	
Basic Flow	**Steps**	
	1	The user selects a choice to be base choice
	Postcondition	A choice is selected as base choice.
Specific Alternative Flows	None	
Global Alternative Flows	None	
Bounded Alternative Flows	None	

Table 22 UC1.2.2 Select Base Choice

Use Case Name	**UC 1.3 Define CP Spec – Output**	
Brief Description	This test case is for defining the output parameters specifications	
Precondition	The user has created a new project	
Primary Actor	User	
Secondary Actor	None	
Dependency	EXTENDED BY USE CASE UC1.5 Verify Parameter Compatibility EXTENDED BY USE CASE UC1.4 Define Parameter INCLUDE USE CASE UC1.3.1 Define Output Equivalence Class	
Generalization	None	
Basic Flow	**Steps**	
	1	The user defines output equivalence classes
	Postcondition	The output equivalence classes are in the system.
Specific Alternative Flows	None	
Global Alternative Flows	None	
Bounded Alternative Flows	None	

Table 23 UC 1.3 Define CP Spec – Output

Use Case Name	UC1.3.1 Define Output Equivalence Class	
Brief Description	This use case is for defining the output equivalence class	
Precondition	The user has selected to define CP specifications-output	
Primary Actor	user	
Secondary Actor	None	
Dependency	EXTENDED BY USE CASE UC1.6 Select Pre-defined Operation EXTENDED BY USE CASE UC1.7 Write JAVA Expression	
Generalization	None	
Basic Flow	Steps	
	1	The user defines the name and data type of an output parameter
	2	The user defines output equivalence class
	Postcondition	The parameter is added to the system
Specific Alternative Flows	BFS2	
	1	The user defines wrong or redundant output equivalence class
	2	The system prompts the user to define correct output equivalence class
	3	The user defines correct output equivalence class.
Global Alternative Flows		
Bounded Alternative Flows		

Table 24 UC1.3.1 Define Output Equivalence Class

Use Case Name	UC1.4 Define Parameter	
Brief Description	The user defines parameters	
Precondition	The user has not loaded TS file	
Primary Actor	User	
Secondary Actor	None	
Dependency		
Generalization	None	
Basic Flow	Steps	
	1	The user defines CP specifications
	2	The user saves the data
	Postcondition	The CP specification is saved in system
Specific Alternative Flows	BFS2	
	1	The user defines wrong CP specifications
	2	System prompts user to enter correct values
	3	The user defines wrong CP specifications
Global Alternative Flows		
Bounded Alternative Flows		

Table 25 UC1.4 Define Parameter

Use Case Name	UC1.5 Verify Parameter Compatibility	
Brief Description	This test case verifies the parameters defined by the user to the ones in the TS file loaded into the system by the user.	
Precondition	The user has defined parameters The user has loaded a TS file	
Primary Actor	User	
Secondary Actor	None	
Dependency	None	
Generalization	None	
Basic Flow	Steps	
	1	Name and type entered by the user is compared to the name and type in the TS file that is loaded by the user.
	2	The parameters entered by the user matches the parameters in the TS file
	Postcondition	The parameters are verified
Specific Alternative Flows	Steps	
	1	Name and type entered by the user is compared to the name and type in the TS file that is loaded by the user.
	2	The parameters entered by the user do not match the parameters in the TS file
	Postcondition	The parameters are not verified
Global Alternative Flows		
Bounded Alternative Flows		

Table 26 UC1.5 Verify Parameter Compatibility

Use Case Name	UC1.6 Select Pre-defined Operation	
Brief Description	This use case is for defining an equivalence class using a pre-defined operation	
Precondition	The user has selected to define equivalence class	
Primary Actor	User	
Secondary Actor	None	
Dependency	None	
Generalization	None	
Basic Flow	Steps	
	1	The user selects an operation from a given list.
	Postcondition	The operation is selected
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 27 UC1.6 Select Pre-defined Operation

Use Case Name	UC1.7 Write Java Expression	
Brief Description	This use case is for defining an equivalence class using a Java Expression	
Precondition	The user has selected to define equivalence class	
Primary Actor	User	
Secondary Actor	None	
Dependency	None	
Generalization	None	
Basic Flow	Steps	
	1	The user enters a Java expression
	2	The system checks that the Java expression syntax is correct
	Postcondition	The Java expression is verified and the expression is added in the CP specifications.
Specific Alternative Flows	BFS2	
	1	The Java expression is not correct
	2	the system prompts the user to enter a correct Java expression
Global Alternative Flows		
Bounded Alternative Flows		

Table 28 UC1.7 Write Java Expression

Use Case Name	UC2 Analyze Test Suite	
Brief Description	This test case is used to describe the process of analyzing problems and suggesting changes.	
Precondition	The user has defined the test suite	
Primary Actor	User	
Secondary Actor	None	
Dependency	INCLUDE USE CASE UC2.1 Build ATS INCLUDE USE CASE UC2.2 Run DT Algorithm INCLUDE USE CASE UC2.3 Analyze Results	
Generalization	None	
Basic Flow	Steps	
	1	The system builds an ATS
	2	The system runs DT Algorithm
	3	The system analyzes results
	Postcondition	The test suite is analyzed by the system
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 29 UC2 Analyze Test Suite

Use Case Name	**UC2.1 Build ATS**	
Brief Description	The system builds an abstract test suite (ATS)	
Precondition	The user chooses to analyze the test suite	
Primary Actor	User	
Secondary Actor	None	
Dependency	INCLUDE USE CASE UC2.1.1 Verify CP Spec Rules	
Generalization	None	
Basic Flow	Steps	
	1	The system builds an ATS
	Postcondition	The ATS is added to the database by the system.
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 30 UC2.1 Build ATS

Use Case Name	**UC2.1.1 Verify CP Spec Rules**	
Brief Description	The system verifies the CP specifications defined by the user	
Precondition	The user has selected to analyze the test suite	
Primary Actor	User	
Secondary Actor	None	
Dependency	None	
Generalization	None	
Basic Flow	Steps	
	1	The system verifies the CP specifications
	2	The CP specifications are correct and they are verified by the system.
	Postcondition	The CP specification is verified
Global Alternative Flows		
Specific Alternative Flows	Steps	
	1	The system verifies the CP specifications
	2	The CP specifications are not correct so the system asks the user to correct the specifications
	Postcondition	The verification is aborted.
Bounded Alternative Flows		

Table 31 UC2.1.1 Verify CP Spec Rules

Use Case Name	UC2.2 Run DT Algorithm	
Brief Description	This use case is for running a decision tree algorithm to find potential problems.	
Precondition	The user selects to analyze the results	
Primary Actor	User	
Secondary Actor	None	
Dependency	EXENDED BY USE CASE UC2.2.1 Report ATS Data Problems	
Generalization	None	
Basic Flow	Steps	
	1	The decision tree algorithm is run
	Postcondition	The decision tree is made by the system for the user to analyze results
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 32 UC2.2 Run DT Algorithm

Use Case Name	UC2.2.1 Report ATS Data Problems	
Brief Description	This use case deals with capturing and reporting Abstract Test Suite(ATS)	
Precondition	Problems are found in the data that is used by ML language.	
Primary Actor	System	
Secondary Actor	None	
Dependency		
Generalization	None	
Basic Flow	Steps	
	1	The problems are reported to the system
	Postcondition	The system shows the problem report to the user.
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 33 UC2.2.1 Report ATS Data Problems

Use Case Name	UC2.3 Analyze Results	
Brief Description	The system analyzes possible root causes for problems	
Precondition	The user selects to analyze test suite.	
Primary Actor	System	
Secondary Actor	None	
Dependency	INCLUDE USE CASE UC2.3.1 Search for Non-conformities	
Generalization	None	
Basic Flow	Steps	
	1	Potential problems found in 2.3.1 is analyzed for root causes
	Postcondition	The potential problems are reported to the user.
Specific Alternative Flows		
Global Alternative Flows		

Bounded Alternative Flows	

Table 34 UC2.3 Analyze Results

Use Case Name	UC2.3.1 Search for Non-conformities	
Brief Description	This involves searching for Non-conformities in the test suite	
Precondition	The user selects to analyze test suite	
Primary Actor	System	
Secondary Actor	None	
Dependency	INCLUDES USECASE UC2.3.2 Analyze Possible Causes	
Generalization	None	
Basic Flow	Steps	
	1	The system searches for any problems in the test suite
	Postcondition	Non-conformities are reported to the user.
Specific Alternative Flows		
Global Alternative Flows	Steps	
	1	The system searches for any problems in the test suite
	Postcondition	No non-conformity problems are reported
Bounded Alternative Flows		

Table 35 UC2.3.1 Search for Non-conformities

Use Case Name	UC2.3.2 Analyze Possible Causes	
Brief Description	This involves analyzing possible causes for the problems identified in 2.3.1	
Precondition	Problems are found in 2.3.1	
Primary Actor	System	
Secondary Actor	None	
Dependency	EXTENDED BY USECASE UC2.3.3Evaluate Possible Corrections	
Generalization	None	
Basic Flow	Steps	
	1	The system finds possible causes of problems found in 2.3.1
	Postcondition	Possible causes are reported to the user
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 36 UC2.3.2 Analyze Possible Causes

Use Case Name	UC2.3.3Evaluate Possible Corrections
Brief Description	This use case involves evaluating possible corrections for problems found.
Precondition	Problems are found
Primary Actor	System
Secondary Actor	None
Dependency	

Generalization	None	
Basic Flow	Steps	
	1	The system evaluates possible corrections for problems found
	Postcondition	The system reports possible corrections to the user.
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 37 UC2.3.3Evaluate Possible Corrections

Use Case Name	**UC3 Modify CP Spec**	
Brief Description	This test case allows the user to change CP specifications	
Precondition	The user selects to change CP specifications	
Primary Actor	User	
Secondary Actor	None	
Dependency	EXTENDED BY USECASE UC 1.2.1 Define Category and Choices EXTENDED BY USECASE UC Define Output Equivalence Class	
Generalization	None	
Basic Flow	Steps	
	1	The user changes a specification or specifications
	2	The user saves the modified specification
	Postcondition	The specification is saved in the data base by the system.
Specific Alternative Flows	BFS	
	1	The user saves the modified specification
	2	The system prompts the user to enter correct values
	3	The user enters correct values
	Postcondition	The specification is saved in the data base by the system.
Global Alternative Flows		
Bounded Alternative Flows		

Table 38 UC3 Modify CP Spec

Use Case Name	**UC4 Modify Test Suite**	
Brief Description	This test case allows the user to modify test suite	
Precondition	The user selects to modify test suite	
Primary Actor	User	
Secondary Actor	None	
Dependency	EXTENDED BY USECASE UC4.1 Define New TC	
Generalization	None	
Basic Flow	Steps	
	1	The user changes the test suite
	2	The user saves the modified test suite
	Postcondition	The test suite is saved in the data base by the system.

Specific Alternative Flows	
Global Alternative Flows	
Bounded Alternative Flows	

Table 39 UC4 Modify Test Suite

Use Case Name	**UC4.1 Define New TC**	
Brief Description	This test case allows the user to define a new test case	
Precondition	The user selects to define TC	
Primary Actor	None	
Secondary Actor	None	
Dependency		
Generalization	None	
Basic Flow	Steps	
	1	The user adds a new test case
	Postcondition	The test case is added to the database by the system
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 40 UC4.1 Define New TC

Use Case Name	**UC5 Persist Data**	
Brief Description	This test case allows the user to save the data	
Precondition	The user selects to save data	
Primary Actor	None	
Secondary Actor	None	
Dependency		
Generalization	None	
Basic Flow	Steps	
	1	The data is saved in the data base by the system
	Postcondition	The data is in database.
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 41 UC5 Persist Data

Use Case Name	UC6 Make Test Suite	
Brief Description	Make test suite is a use case used for building a test suite using CP specifications	
Precondition	User has defined CP Specification User selects MakeTestSuite	
Primary Actor	User	
Secondary Actor	None	
Dependency		
Generalization	None	
Basic Flow	Steps	
	1	User defines CP specs –inputs and outputs and coverage criteria
	2	System makes test suite using specifications
	Postcondition	Test suite is constructed
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 42 UC6 Make Test Suite

Use Case Name	UC6.1 Select Coverage Criterion	
Brief Description	This use case is used for the user to select a coverage criterion from a given list.	
Precondition	The user has asked the system to make a test suite.	
Primary Actor	User	
Secondary Actor	None	
Dependency		
Generalization	None	
Basic Flow	Steps	
	1	The user selects from a given list of criteria which is: All Coverage, Base Choice, Each Choice and Pair wise.
	Postcondition	The criterion is selected.
Specific Alternative Flows		
Global Alternative Flows		
Bounded Alternative Flows		

Table 43 UC6.1 Select Coverage Criterion

APPENDIX B TRIANGLE CASE STUDY

B1. CP Specification

We have three inputs; therefore we have three parameters called SA, SB and SC. Each of these parameters has three categories. The first category of each parameter (i.e. Cat1, Cat4 and Cat7) is to verify whether the input integer is strictly greater than 0 or not: in the latter case, then it is an error, whereas the former case corresponds to the normal mode of execution (Base Choice). Cat2 is to compare if the value of SA is equal to SB or not, i.e., whether the triangle is (at least) isosceles with these two sides. This category is similar to Cat5 and Cat8. These three categories can be used to determine the type of the triangle: e.g., if we have ch2.1, ch5.1, and ch8.1 the triangle is equilateral. Cat3, Cat6 and Cat9 are used to determine if we have a triangle or not, i.e., if the triangle inequalities hold. We decided that each situation that does not lead to a triangle, i.e., ch6, ch12, and ch18, would be exercised only once, thus the constraint [single].

There are three constraints for ch8.1 and ch8.2. If we have SA equal to SB (i.e., ch2.2) and SB equal to SC (i.e., ch5.1), by transitivity SA is equal to SC (i.e., ch8.1), therefore we cannot have combinations of ch2.2, ch5.1 and ch8.2 and we have added this constraint to ch8.2. Also by the same rule of transitivity, if SA does not equal to SB (i.e., ch2.3) and SB equals to SC (i.e., ch5.1, then SA does not equal to SC (i.e., ch8.2), and also, if SA equals SB (i.e., ch2.2) and SB does not equal to SC (i.e., ch5.2), then SA does not equal to SC (i.e., ch8.2). These two constraints are added to ch8.1. These rules are summarized below:

1. If (SA==SB) && (SB==SC) then SA==SC, and therefore SA!=SC is not possible;
2. If(SA != SB) && SB==SC) then SA!=SC, and therefore SA==SC is not possible;
3. If(SA==SB && SB!=SC) then SA!=SC, and therefore SA==SC is not possible.
4. If(SA!=SB && SB!=SC) then SA!=SC, and therefore SA==SC is not possible.

There are a few rules that apply to ch3.2, ch6.2 and ch9.2:

1. If (SA>=SB+SC) then SA!=SB and SA!=SC (since ch4.2 and ch7.2 specify that SB<=0 and SC<=0 and are error choices, when ch3.2 is used in a frame, SB and SC are strictly

positive), and therefore SA==SB (ch2.1) and SA==SC (ch8.1) cannot be combined with SA>=SB+SC (ch3.2).

2. If (SB>=SC+SA) then SB!=SC and SB!=SA (since ch7.2 and ch1.2 specify that SC<=0 and SA<=0 and are error choices, when ch 6.2 is used in a frame, SC and SA are strictly positive), and therefore SB==SC (ch5.1) and SB==SA (ch2.1) cannot be combined with SB>=SC+SA (ch6.2).

3. If (SC>=SA+SB) then SC!=SA and SC!=SB, (since ch1.2 and ch4.2 specify that SA<=0 and SB<=0 and are error choices, when ch 9.2 is used in a frame, SC and SA are strictly positive) and therefore SC==SA and SC==SB cannot be combined with SC>=SA+SB.

4. If (SC>=SA+SB) then SB<SC+SA and SA<SB+SC, and therefore SA>=SB+SC or SB>=SC+SA cannot be combined with SC>=SA+SB.

Parameters	Categories	Choices	Properties	Selectors
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]
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 && P81)] [single]
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 !(P21 && P52) && !(P22 && P52) && !(P22 && P51)
		[ch8.2] SC != SA	P82	[if !(P21 && P51) [Base Choice]
	[Cat 9] SC compared to SA and SB	[ch9.1] SC < SA + SB		[Base Choice]
		[ch9.2] SC >= SA + SB	P92	[if !(P51 && P81)] [single]

Table 44 Triangle Case Study: CP Specification

B2. Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=23, SB=35, SC=21	hasValidValues=true Type=Scalene
2	C 1.1 |C 2.1 |C 3.1 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=25, SB=25, SC=21	hasValidValues=true Type=Iscosceles
3	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.1| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=123, SB=134, SC= 134	hasValidValues=true Type=Iscosceles
4	C 1.1 |C 2.2 |C 3.2 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=123, SB= 32, SC= 21	hasValidValues=true Type=NotATriangle
5	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2| C 6.2 |C 7.1 |C 8.2 |C 9.1 |	SA=12, SB= 29, SC= 13	hasValidValues=true Type=NotATriangle
6	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.2 |	SA=43, SB= 39, SC= 90	hasValidValues=true Type=NotATriangle
7	C 1.2 |	SA=0, SB= 1, SC= 2	hasValidValues=false Type=NotATriangle
8	C 4.2 |	SA=1, SB= 0, SC= 3	hasValidValues=false Type=NotATriangle
9	C 7.2 |	SA=1, SB= 2, SC=0	hasValidValues=false Type=NotATriangle

Table 45 Triangle Case Study: Base Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.1 |C 3.1 |C 4.1 |C 5.1 |C 6.1 |C 7.1 |C 8.1 |C 9.1 |	SA=21, SB=21, SC=21	hasValidValues=true Type=Equilateral
2	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2 |C 6.2 |C 7.1 |C 8.2 |C 9.1 |	SA=21, SB=45, SC=21	hasValidValues=true Type=NotATriangle
3	C 1.1 |C 2.2 |C 3.2 |C 4.1 |C 5.1 |C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=50, SB=23, SC=23	hasValidValues=true Type=NotATriangle
4	C 1.1 |C 2.1 |C 3.1 |C 4.1 |C 5.2 |C 6.1 |C 7.1 |C 8.2 |C 9.2 |	SA=23, SB=23, SC=58	hasValidValues=true Type=NotATriangle
5	C 1.2 |	SA=0, SB=1, SC=2	hasValidValues=false Type=NotATriangle
6	C 4.2 |	SA=1, SB=0, SC=3	hasValidValues=false Type=NotATriangle
7	C 7.2 |	SA=1, SB=2, SC=0	hasValidValues=false Type=NotATriangle

Table 46 Triangle Case Study: Each Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.1 |C 3.1 |C 4.1 |C 5.1| C 6.1 |C 7.1 |C 8.1 |C 9.1 |	SA=1, SB=1,SC=1	hasValidValues=true Type=Equilateral
2	C 1.1 |C 2.1 |C 3.1 |C 4 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=2, SB=2, SC=3	hasValidValues=true Type=Isosceles
3	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=1, SB=2, SC=3	hasValidValues=true Type=Scalene
4	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.1| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=1, SB=2, SC=2	hasValidValues=true Type=Isosceles
5	C 1.1 |C 2.2 |C 3.2 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.1 |	SA=45, SB=11, SC=21	hasValidValues=true Type=NotATriangle
6	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2| C 6.2 |C 7.1 |C 8.2 |C 9.1 |	SA=11, SB=45, SC=21	hasValidValues=true Type=NotATriangle
7	C 1.1 |C 2.2 |C 3.1 |C 4.1 |C 5.2| C 6.1 |C 7.1 |C 8.2 |C 9.2 |	SA=21, SB=11, SC=45	hasValidValues=true Type=NotATriangle

8	C 1.2 |	SA=0, SB=1, SC=2	hasValidValues=false Type=NotATriangle
9	C 4.2 |	SA=1, SB=0, SC=1	hasValidValues=false Type=NotATriangle
10	C 7.2 |	SA=1, SB=1, SC=0	hasValidValues=false Type=NotATriangle

Table 47 Triangle Case Study: Pair Wise Test Frames

B3. Explanation of Mutants Killed or Alive

There were a total of nine mutants generated for class level mutants. The first three mutants are JSI_1 to JSI_3, where the program changes the input parameters from int value to static int. The values of input parameters are not changed anywhere in the program therefore changing the value to static has no impact on the variables. Since this change does not have any impact on the functionality of the program, these mutants are not killed by any of the test suites. These mutants are called equivalent mutants.

The next set of mutants is JTD_1 to JTD_3. The constructor in Triangle class uses the same name for inputs given by the user and instance variables of the class. Removing `this` keyword means assigning the input variable to itself, as opposed to assigning the input variable to the instance variable of the class. This results in the mutant being killed by all test suites.

Mutants JTI_1 to JTI_3, insert `this` keyword in the java program. By inserting `this` keyword in the constructor, (i.e. changing `this.a = a` to `this.a = this.a`) the value of the class variable is set to null, which is its initial value. This will cause the type of triangle returned to be `NotATriangle` for any value of input variables. Therefore these mutants are killed by all test suites.

AORB mutants generated refer to the following part of the code, which is used to determine if we have a triangle:

```
if (a <= b + c && b <= a + c && c <= a + b){
        //determine the type of triangle
}
Else{
        //not a triangle
}
```

It is effortless for the test suits to kill the AORB mutants, because these mutants are changing the definition of a triangle. An example is changing the statement: `if (a <= b + c && b <= a + c && c <= a + b)` to `if (a <= b + c && b <= a + c && (c <= a / b)`.

Changing the binary operator means that no input value satisfies the mutant condition and therefore these mutants are killed by all test suites.

AORB_1 to AORB_4 change the + operator in `a <= b + c` to *, /, % and -.

AORB_1 is killed with an input satisfying `b+c>a` but `b*c<a` or vice versa.

AORB_2 is killed with an input satisfying `b+c>a` but `b/c<a` or vice versa.

AORB_3 is killed with an input satisfying `b+c>a` but `b%c<a` or vice versa.

AORB_4 is killed with an input satisfying `b+c>a` but `b-c<a` or vice versa.

Since all the test suites contain such test cases, all the test suites kill the mutants. Killing these mutants is therefore due to the selection of test inputs and different test inputs satisfying the same test frames may have left the mutants alive.

There are three AOIU mutants created and all these are killed by all three test suites. These mutants are created for the Triangle constructor. When changing the values of parameters from positive to negative, they no longer define a triangle, resulting in the mutants being very easy to kill.

AOIS_1 to AOIS_12 involve changing the constructor which has the following syntax:

```
public Triangle( int a, int b, int c )
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }
```

When a shortcut operator is added before a variable and assigned to another variable (i.e. `this.a = ++a`) the value of `a` is incremented/decremented then assigned to `this.a`. On the other hand, if the shortcut operator is added after a variable and assigned to another variable (i.e. `this.a = a++`), the value is assigned then incremented. Mutants AOIS_1, 2, 5, 6, 9 and 10 use shortcut operators before the variable, and since this changes the value right away (i.e. `a = 3` instead of 2) impacting the type of triangle, these mutants are killed by all test suites. AOIS_3, 4, 7, 8, 11 and 12 however have shortcut operators after the variable. Since `a`, `b`, and `c` are not used further

in the code, these mutants become equivalent and are therefore not killed by any of the test suites.

AOIS_13 to AOIS_24 change the first line in the following piece of code:

```
if (a > 0 && b > 0 && c > 0) {
        return true;
    } else {
        return false;
    }
```

This part of the code determines if the input parameters are valid values. Since all test suites contain a test case that contains zero values for each of the variables, these mutants are easy to detect and kill. An example is changing `a>0` to `a++>0`. Since we are incrementing the value of `a` and comparing it to 0, this statement returns true where it should return false. This is why these mutants are killed by all test suites.

AOIS_25 to AOIS_60 involve changing the statement:

```
if (a <= b + c && b <= a + c && c <= a + b)
```

Since there is at least one test case whereby adding an operator before or after the variables in this statement causes a change in the return value, these mutants are killed by all test suites.AOIS_61 to AOIS_76 involve changing the statement:

```
if (a == b && b == c) {
                    return "Equilateral";
                }
```

AOIS_61, AOIS_62, and AOIS_65 to AOIS_70 are killed by all test suites.

AOIS_61 to AOIS_64 add shortcut operators to a. Since AOIS_61 and AOIS_62 add the shortcut operator before a, the value of a is incremented/decremented and compared to b. This statement will result in a not equal to b even if they are equal indeed. Therefore these two mutants are killed by all test suites.

AOIS_63 and AOIS_64 add the shortcut operator after `a`. This change results in the value of `a` changing after the comparison is done. Changing the value of `a` after comparison has no effect on the result when there is an `Equilateral` triangle. If there is a triangle that is not `Equilateral`, the program starts by making this comparison and the program exits the check by incrementing the value of `a`. If the test case exercises `NotATriangle`, this part of the code

is never reached. Since the Each Choice test suite does not have any test case other than `Equilateral` and `NotATriangle`, these mutants are never killed. Because of the nature of the CP specification, changing the values of the test cases will not kill this mutant.

Incrementing the value of `a` can have an impact on the other test cases. The Base Choice test suite has a test frame where the value of `a` is equal to `b` and the triangle is `isosceles`. Since incrementing/decrementing the value of `a` changes the triangle from `isosceles` to `Scalene` (25, 25, 21 would change into 24, 25, 21 for instance), these mutants are caught by the Base Choice test suite. The Pair Wise test suite also has a test case where changing the value of `a` has an impact on the type of triangle (for instance 1, 2, 3). Incrementing the value of `a` causes the triangle to be `isosceles` (2, 2, 3), and decrementing it causes the triangle to be `NotATriangle` (0, 2, 3). Therefore these are caught by the Pair Wise test suite as well.

AOIS_65 to AOIS_68 add shortcut operators to `b` in the first check in the statement. Since there is a check for values of `b` equal to `c` in the same line, these mutants are killed by all test suites regardless of the operation being before or after the variable.

AOIS_69 and AOIS_70 are killed by all mutants because of the same explanation provided for AOIS_61 and AOIS_62.

AOIS_71and AOIS_72 are not killed by the Each choice test suite because of the reason provided for AOIS_63 and AOIS_64. AOIS_71 is not killed by the Pair Wise test suite either. The reason for the mutant to be alive is because the test cases in the Pair Wise test suite will result in the same type of triangle when `b` is incremented and when `a` does not equal to `b` the next statement (`b==c`) is not checked.

AOIS_73 adds plus shortcut operator before `c` in the statement. Since incrementing `c` does not have any impact on any of the test cases in the Base Choice test suite, this mutant is not caught. The reason for no impact is that the test cases that reach this statement and increment `c` do not change the triangle type. An example is having 25, 25, 21 for values of `a`, `b` and `c`. When reaching this statement since `a` equals to `b`, the mutant checks if `b` is equal to `c` and increments the value of `c` to 22. Changing the value to 22 still makes the triangle to be `isosceles`. If the

values for this test frame where chosen differently (i.e. c =24) this mutant would have been killed by the Base Choice test suite as well. The same explanation applies to AOIS_74.

Mutants AOIS_75 and AOIS_76 were not killed by any of the test suites. The value of c is changed after the comparison and since there are no test cases that are impacted by this change this mutant is not caught. The mutants are equivalent.

AOIS_77 to AOIS_99 involve changing the following statement:

```
if (a != b && a != c && b != c) {
return "Scalene";
}
```

Since the Each Choice test suite does not have any test case reaching the *if* statement (because it only has test cases defining `Equilateral` and `NotATriangle` triangles), none of the mutants in this category are killed.

AOIS_77 to AOIS_78 are killed by both the Base Choice and Pair Wise test suites. The value of a changes before comparing and since both test suites have test cases where a is equal to b, these mutants are caught.

AOIS_79 and AOIS_80 are not killed by any of the test suites. Changing the value of a after comparison does not have any impact on the type of triangle because a is never used after this comparison. These mutants are therefore equivalent and not killed.

AOIS_81 to AOIS_84 are killed by both the Base Choice and Pair Wise test suites. Since changing the value of b in the first statement impacts the comparison in the next statements, these mutants are caught by all test suites.

AOIS_85 to AOIS_88 changes the value of a in the second statement where a is being compared to c. There are two reasons for these mutants not to be caught. The mutants which add the shortcut operator before a (thereby changing the value of a before comparison), are not killed because there are no test cases where a does not equal to b and a+1 is equal to c. The mutants in this category that change the value of a after comparison are not killed because the value of a is not used later on in the line. These mutants are equivalent.

Mutants AOIS_89 to AOIS_92 are killed by both the Base Choice and Pair Wise test suites. These mutants add shortcut operator to c in the second part of the statement. Since c is used again to compare against b in the next part, it has an impact on the return statement. Therefore these mutants are killed by both test suites.

AOIS_93 to AOIS_96 involve changing b in the third part of the statement. AOIS_93 and AOIS_94 change the value of b when comparing, therefore these are caught and killed. However AOIS_95 and AOIS_96 are not killed by the test suites since the change the value of b after the comparison and since this value is never used again by the program it does not have any impact on the return value. These mutants are equivalent. The same explanation applies to AOIS_97 to AOIS_99, but this time adding shortcut values to c as opposed to b.

ROR_1 to ROR_15 correspond to the following part of the code:

```
if (a > 0 && b > 0 && c > 0) {
            return true;
        }
```

ROR_1 to ROR_4, ROR_6 to ROR_9, and ROR_11 to ROR_15 are killed by all test suites. The reason for these mutants being killed is that the boundary values for a, b and c are tested in a test case by each of these test suites. The only mutants not killed in this set are ROR_5, ROR_10 and ROR_15. These mutants change `a > 0, b > 0` and `c > 0` to `a != 0, b != 0` and `c != 0`, respectively. There are test cases in each test suite involving a, b or c equal to 0, but there are no test cases with these values being less than 0. Therefore in this case `a !=0` is equivalent to saying `a>0`. If we add a test case where the values are smaller than 0, these mutants will be killed.

ROR_16 to ROR_30 involve changing the relational operators in the following statement:

```
if (a <= b + c && b <= a + c && c <= a + b)
```

Changing the relational operators (except for `<= to <`) changes whether the statement returns true or not for at least one test case in each of the test suites. Therefore all the mutants but ROR_18, ROR_23, and ROR_28 are killed by all test suites.

ROR_18, changes `a<=b+c` to `a<b+c` and ROR_23 changes `b<=a+c` to `b<a+c`. Since there are no test cases where `a=b+c` or `b=a+c`, these mutants are not killed by any of the test suites.

ROR_28 changes c<=a+b to c<a+b, since the Pair Wise test suite contains a test case where c=a+b, this mutant is killed. However the Base Choice and Each Choice test suites do not have any such test cases therefore they are unable to kill these mutants. In this case by changing the values for a test frame in the Base Choice (e.g., Scalene (1,2,3)) these mutants can be killed by the Base Choice test suite as well. However the Each Choice test suite cannot accommodate such test case because of the nature of its test frames.

ROR_31 to ROR_40 involve changing the relational operators in the following statement:

```
if (a == b && b == c) {
                    return "Equilateral";
                }
```

ROR_33, ROR_35 to ROR_36 and ROR_40 are killed by all test suites. These mutants change the == operator in the first statement to < and !=, and == operator in the second statement to > and !=. The mutants are killed because with these conditions the value returned is not Equilateral when it should be in the case of the Each Choice test suite, the value returned is an Equilateral when it should not be (i.e. 1,2,2 is Isosceles but returned value is Equilateral) in the case of the Pair Wise and Base Choice test suites.

ROR_31 changes a==b to a>b. If the triangle is Equilateral, this statement will return false and the type of triangle will be changed, therefore this mutant will be killed if there is an Equilateral triangle. If a is greater than b and b is equal to c, this will be an invalid triangle therefore there are no other test cases that can kill this mutant. Since the Base Choice test suite does not contain a test case with Equilateral triangle, this mutant is not killed by this test suite. Therefore ROR_31 is not killed by the Base Choice test suite.

ROR_32 is not killed by any of the test suites. The mutant changes a==b to a>=b. This mutant would be killed if the value of a were greater than b and b were equal to c. But this test case is impossible since these values would not make a triangle. Therefore this mutant is equivalent and not killed by any of the test suites.

ROR_41 to ROR_55 involve changing the relational operators in the following statement:

```
if (a != b && a != c && b != c) {
     return "Scalene";
    }
```

ROR_43 changes `a!=b` to `a<b` and is not killed by any of the test suites. The reason for this mutant to be alive is that `a<b` means that `a` does not equal to `b`, and there are no other cases where `a` is less than `b` and is not `Scalene`.

All the other mutants in this category are killed by the Base Choice and Pair Wise test suites. Since there is no input value that satisfies the condition, these mutants are killed.

COI_1 to COI_5 and COR_1 to COR_4 involve the following line of code:

```
if (a > 0 && b > 0 && c > 0) {
            return true;
    }
```

All these mutants are killed by all test suites, the reason is that all test suites involve test cases where there is a check for whether the inputs are valid values or not. By adding or removing conditional operators to any of the statements the statement does not return true anymore. Therefore all these mutants are killed.

COI_6 to COI_11 and COR_5 to COR_8 involve the following line of code:

```
if (a <= b + c && b <= a + c && c <= a + b)
```

This part of the code determines if we have a triangle or not. Negating any of the expressions i.e. `a<=b+c` to `!(a<=b+c)` will cause the statement to return true instead of false and vice versa. An example is if we have a valid triangle, then `a<=b+c` would be true. Changing this to `!(a<=b+c)` makes this statement return false and the expression evaluates to false. The test frames allow for at least one test case to have valid triangle values. Since all test suites contain a test case with valid triangle values, these mutants are killed by all of them.

COI_12 to COI_13 and COR_9 and COR_10 involve the following line of code:

```
if (a == b && b == c) {
                    return "Equilateral";
                }
```

Inserting or removing the conditional operators to this statement changes the behavior. Since the Each Choice and Pair Wise test suites have test cases with `Equilateral` triangle, these mutants are killed by them. Although the Base Choice test suite does not contain any test case with `Equilateral` triangle, the mutants are killed because this statement returns true when indeed it should return false.

COR_10 is killed by all test suites since all test suites contain a test case for `Equilateral` triangle and in this case the `&&` operator and bit wise exclusive or operator have different functionalities therefore if both conditions are met the statement returns false.

COI_14 to COI_18 and COR_11 to COR_14 involve changing the first line of the following code:

```
if (a != b && a != c && b != c) {
                        return "Scalene";
                    }
```

These mutants are killed by the Base Choice and Pair Wise test suites but not by the Each Choice. The Each Choice test suite does not have any test case that involves a `Scalene` triangle therefore these mutants are never found.

LOI_1 to LOI_3 involve the following lines of code:

```
public Triangle( int a, int b, int c )
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }
```

Since these mutants add a bitwise operator to the integer (i.e. `this.a=~a`), the value assigned to the variables of the function are negated, making the output value to be `NotATriangle` when it is a triangle. Therefore they are caught by all test suites.

LOI_4 to LOI_6 involve the following lines of code:

```
if (a > 0 && b > 0 && c > 0) {
            return true;
        }
```

These mutants are killed by all test suites because of the same reason provided for LOI_1 to LOI_3.

LOI_7 to LOI_15 involve the following line of code:

```
if (a <= b + c && b <= a + c && c <= a + b) {
```

Inserting the bitwise operators to `a`, `b` and `c` on the left hand side causes the program to return true when it is false. An example is changing `a` to `~a` in the first statement. If we have values (4,1,1), this is not a triangle. The value of 4 is changed to -4 causing the statement to return true as opposed to false. Changing the statements on the right hand side of the triangle leads the right

hand side to be smaller in some cases (if the negative value is greater or both values are equal). If the test case is `Equilateral` these mutants will be killed since the right hand side will always equal to 0 and the left hand side will be greater than right hand side. Since all test suites have a test case with the above-mentioned description, all these mutants are killed by all test suites. These mutants are killed by each test suite because of the construction of test frames (i.e., any input satisfying the conditions of the test frames would kill the mutants).

LOI_16 to LOI_19 involve the following line of code:

```
if (a == b && b == c) {
                    return "Equilateral";
                }
```

These mutants are killed by the Each Choice and Pair Wise test suites but not by the Base Choice test suite. The reason behind this is that there are no `Equilateral` triangle test cases in the Base Choice test suite. These mutants will be caught if a test case was added to the suite where the triangle is `Equilateral`.

LOI_20 to LOI_25 involve the following line of code:

```
if (a != b && a != c && b != c) {
                        return "Scalene";
                    }
```

Since the Each Choice test suite does not contain any test case that is a `Scalene` triangle, these mutants are never caught.

LOI_22 and LOI_23 change the values of `a` and `c` (i.e. `~a` and `~c`) in the second part of the statement. Since there are no test cases where `a` is equal to `c`, these mutants are not killed by any of the test suites.

The other mutants are killed by the Base Choice and Pair Wise test suites because these two test suites contain triangles that are `Scalene` and also test cases where `a` is equal to `b` or `b` is equal to `c` (`isosceles`).

The triangle program does not contain AORU, AODU, AORS, AODS, COD, COR, COI and COD mutants because of the specifics of the code.

APPENDIX C NEXTDATE CASE STUDY

C1.CP Specification

To model this program using category partitioning, we start by identifying the parameters, which in this case are the inputs year, month, and day. The next step is to identify the categories and choices for each parameter.

The categories of the first parameter (year) are as follows. Since a year is between 1800 and 2012, we need to test whether the input falls into this range or outside this range, therefore we would have a category for the range of the year. This category would have three choices; one is when we have a year before 1582, another choice with years falling inside range [1582, 2100], which we would choose as a Base Choice, and the third choice which falls after the range, meaning any year after 2100. The first and third choices in this category are error choices since they are out of range. The year parameter would also have another category to determine if the year is a leap year or not. Therefore there would be two choices, one is for leap year and one is for a common year, the latter one being the Base Choice.

The next parameter is the month and the categories are described as follows. Since there are months with 31 or 30 days, and February has either 28 or 29 days, we would define a category with the number of days in a month, and the choices would be: 31 days, 30 days, or February. Even though December has 31 days, we would like to have a different choice just for December since it is a marginal month, as we need to check that the year is incremented when we switch from December to January. There were also two other choices added, one is when the month number is less than 1, indicating an incorrect value, and when the value is greater than 12, since we only have 12 months in a year. Both of these choices are considered as error choices.

Day is the last parameter and we define its' categories as follows. There will be a category for the number of days, which has six choices. The first choice is to check if the day falls in the range [1-28], which is a valid date for any month. The next three choices are according to end dates of each month, meaning that when these dates are reached we may need to increment the month. For example if we have 29 and it is a leap year and it is February we need to increment the month and go to March. The same applies to values 30 and 31 for other months in the year. Therefore we have choices for 29, 30 and 31. We would also want to test if the user enters the

wrong date, meaning a number that is greater than 31 or less than 1, which leads to two (error) choices.

The following table shows the category partition for the next date problem.

Parameters	Categories	Choices	Properties	Constraints
Year	**[Cat 1]** Range	**[ch1.1]** Year<=1582		[error]
		[ch1.2] 1582<=Year<=2100		[Base Choice]
		[ch1.3] Year>2100		[error]
	[Cat 2] Type	**[ch2.1]** isLeapYear		
		[ch2.2] isCommonYear	[isCommon]	[Base Choice]
Month	**[Cat 3]** Length	**[ch3.1]** 30Days (4, 6, 9, 11)	[has30Days]	
		[ch3.2] 31Days(except December) (1, 3, 5, 7, 8, 10)		[Base Choice]
		[ch3.3] February (2)	[isFeb]	
		[ch3.4] December (12)	[isDec]	
		[ch3.5] month<1		[error]
		[ch3.6] month>12		[error]
Day	**[Cat 4]** Range	**[ch4.1]** 1<=Day<=28		[Base Choice]
		[ch4.2] Day<1		[error]
		[ch4.3] Day>31		[error]
		[ch4.4] Day=29		[If !(isCommon&&isFeb)]
		[ch4.5] Day=30		[If !isFeb]
		[ch4.6] Day=31		[If !isFeb && !has30Days]

Table 48 NextDate Case Study: CP Specification

Choices ch1.2, ch2.2, ch3.2 and ch4.1 have been chosen to be the Base Choices for these categories. Choices ch1.2, ch2.2 and ch4.1 are selected as Base Choices because these are the most common dates that are used in the calendar and they would be combined with any other choices from other categories to make a valid date (except for error choices). Ch3.2 is also used as a Base Choice, since there are more months with 31 days than there are months with 30.

C2.Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.2 |C 3.2 |C4.1 |	1986,1,25	1986,1,26
2	C 1.2 |C 2.1 |C 3.2 |C 4.1 |	1984,7,1	1984,7,2
3	C 1.2 |C 2.2 |C 3.1 |C 4.1 |	1983,9,18	1983,9,19
4	C 1.2 |C 2.2 |C 3.3 |C 4.1 |	1994,2,16	1994,2,17
5	C 1.2 |C 2.2 |C 3.4 |C 4.1 |	1857,12,21	1857,12,22
6	C 1.2 |C 2.2 |C 3.2 |C 4.4 |	2001,3,29	2001,3,30
7	C 1.2 |C 2.2 |C 3.2 |C 4.5 |	1743,10,30	1743,10,31
8	C 1.2 |C 2.2 |C 3.2 |C 4.6 |	1609,5,31	1609,6,1
9	C 1.1 |	1502,1,1	Null
10	C 1.3 |	3000,1,2	Null
11	C 3.5 |	2010,0,1	Null
12	C 3.6 |	2010,13,2	Null
13	C 4.2 |	2010,3,0	Null

14	C 4.3 |	2010,3,38	Null

Table 49 NextDate Case Study: Base Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.2 |C 3.3 |C 4.1 |	1986,2,28	1986,3,1
2	C 1.2 |C 2.1 |C 3.2 |C 4.4 |	1984,3,29	1984,3,30
3	C 1.2 |C 2.1 |C 3.1 |C 4.5 |	1708,9,30	1708,10,1
4	C 1.2 |C 2.1 |C 3.1 |C 4.6 |	1592,12,31	1593,1,1
5	C 1.1 |	3000,1,1	Null
6	C 1.3 |	2010,0,1	Null
7	C 3.5 |	2010,13,2	Null
8	C 3.6 |	2010,3,0	Null
9	C 4.2 |	2010,3,38	Null
10	C 4.3 |	1506,1,1	Null

Table 50 NextDate Case Study: Each Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.2 |C 3.1|C 4.4 |	1831,4,29	1831,4,20
2	C 1.2 |C 2.2 |C 3.3 |C 4.1 |	1759,2,28	1759,3,1
3	C 1.2 |C 2.1 |C 3.3 |C 4.4 |	1616,2,29	1616,3,1
4	C 1.2 |C 2.1 |C 3.1 |C 4.1 |	1724,6,27	1724,6,28
5	C 1.2 |C 2.1 |C 3.4 |C 4.4 |	1888,12,29	1888,12,30
6	C 1.2 |C 2.1 |C 3.1 |C 4.5 |	1896,11,30	1896,12,1
7	C 1.2 |C 2.2 |C 3.4 |C 4.5 |	1971,12,30	1971,12,31
8	C 1.2 |C 2.2 |C 3.2 |C 4.1 |	1571,8,25	1571,8,26
9	C 1.2 |C 2.1 |C 3.2 |C 4.4 |	1744,10,29	1744,10,30
10	C 1.2 |C 2.1 |C 3.2 |C 4.6 |	1736,1,31	1736,2,1
11	C 1.2 |C 2.1 |C 3.2 |C 4.5 |	2012,5,30	2012,5,31
12	C 1.2 |C 2.1 |C 3.4 |C 4.1 |	2000,12,28	2000,12,29
13	C 1.2 |C 2.2 |C 3.4 |C 4.6 |	2031,12,31	2032,1,1
14	C 1.1 |	3000,1,1	Null
15	C 1.3 |	2010,0,1	Null
16	C 3.5 |	2010,13,2	Null
17	C 3.6 |	2010,3,0	Null
18	C 4.2 |	2010,3,38	Null
19	C 4.3 |	1506,1,1	Null

Table 51 NextDate Case Study: Pair Wise Test Frames

C3.Explanation of Mutants Killed or Alive

There is one class level mutant generated for the next date problem which is IPC_1. Removing the `super( year, month, day )`, statement from the constructor means that the default values of year, month and day are set to zero. This means that the next date is zero as well, making the actual value equal to the test value. Therefore this mutant is not killed by any of the test suites.

AOIS_1 to AOIS_6 involve the following lines of code:

```
public NextDate( int year, int month, int day )
    {
        super( year, month, day );
    }
```

Since these mutants make changes to the constructor, the same changes apply to the object that is created in the test case to compare against. This is why these mutants are not killed. These mutants are equivalent.

LOI_1 to LOI_3 and AOIU_1 to AOIU_3 are killed by all test suites, because the value of the parameters are negated and when there is an invalid value (-1986) the value returned is 0,0,0. When the value is compared to the correct value (even though this is the wrong value), the values are not the same. Therefore these mutants are killed.

AORB_1 to AORB_4 correspond to the following part of the code:

```
case 10:
if (this.getDay() < 31) {
                    this.setDay( this.getDay() + 1 );
                }
```

Since all test suites have a test case where the value of day is less than 31 and changing the addition returns a wrong value for day parameter, these mutants are killed by all test suites.

AORB_5 to AORB_8 correspond to the following part of the code:

```
else {
                    this.setDay( 1 );
                    this.setMonth( this.getMonth() + 1 );
                }
```

Since the Each Choice test suite does not contain any test cases where the month is a 31 day month and the day is 31, these mutants are not killed by this test suite. However these mutants are killed by the Base Choice and Pair wise test suites.

AORB_9 to AORB_12 correspond to the following part of the code:

```
case 11 :
                if (this.getDay() < 30) {
                    this.setDay( this.getDay() + 1 );
                }
```

There are no test cases in test suite that reach this part of the code (i.e. no test cases that the month has 30 days and the day is less than 30). Therefore these mutants are not killed by this test

suite. Since the Base Choice and Pair Wise test suites contain a test case with the above mentioned criterion, they are able to kill these mutants.

AORB_13 to AORB_16 correspond to the following part of the code:

```
else {
      if (this.getDay() == 30) {
        this.setDay( 1 );
        this.setMonth( this.getMonth() + 1 );
            }
```

No test cases correspond to this part of the code in the Base Choice test suite. Therefore these mutants are not killed. However the Each Choice and Pair Wise test suites contain a test case which covers this portion, therefore these mutants are killed by it.

AORB_17 to AORB_AORB_20 correspond to the following part of the code:

```
case 12 :
                if (this.getDay() < 31) {
                    this.setDay( this.getDay() + 1 );
                }
```

These mutants are killed by the Pair Wise and Base Choice test suites, since they contain a test case that reaches this part of the code. The Each Choice test suite, however, does not contain such test case and is therefore not able to kill these mutants.

AORB_21 to AORB_24 correspond to the following part of the code:

```
else {
this.setYear( this.getYear() + 1 );
      }
```

There are no test cases in the Base Choice test suites that cover this part of the code. (i.e. no test cases covering the case where the month is equal to 12 and the day is not less than 31. Therefore these mutants are not caught by the Base Choice test suite. The Each Choice and Pair Wise test suites however contain a choice with these options. Therefore they are killed.

AORB_25 to AORB_28 correspond to the following part of the code:

```
case 2 :
                if (this.getDay() < 28) {
                    this.setDay( this.getDay() + 1 );
                }
```

There are no test cases involving this part of the code in the Each Choice and Pair Wise test suites, therefore these mutants are not killed by these test suites. The Base Choice test suite however, kills all these mutants. By adding a choice to the CP specification where day is equal to

28, these mutants can be killed by the Pair Wise test suite. The Each Choice test suite however may not kill these mutants since it is not guaranteed that these two choices will appear in the same test frame (i.e. February 28).

AORB_29 to AORB_32 correspond to the following part of the code:

```
if (this.getDay() == 28) {
                        if (this.isLeapYear() == true) {
                            this.setDay( this.getDay() [redacted] 1 );
                        }
```

No test cases in the any of the test suites cover this part of the code. Therefore none of the test suites kill these mutants.

ROR_1 to ROR_5 correspond to the following part of the code:

```
if (this.getYear() < 1508 || this.getYear() [redacted] 2100) {
```

There are no test cases in any of the test suite that have the value of year equal to 2100. Since ROR_1 changes the operator in the second statement from >= to >, this has no impact in the test cases and this mutant is not killed by any of the test suites. To kill this mutant the error choice can be changed to the year 2100.

ROR_2 is killed easily by all test suites, since the operator >= in the second statement is changed to < and this makes the statement return true where it should be false.

ROR_3 is killed by all test suites as well, because of the same reason as ROR_2, except this time the value of >= is changed to <=, which does not make a difference in this case.

ROR_4 and ROR_5 change >= to == and !=. These mutants are killed by all test suites because they all contain an error choice of year greater than 2100 and year not equal to 2100 and not valid.

ROR_6 to ROR_10 correspond to the following part of the code:

```
if (this.getMonth() < 1 || this.getMonth() [redacted] 12) {
            System.out.println( "Invalid month" );}
```

Since all test suites contain a test case with value of month equal to 12 and less than 12, these mutants are killed by all test suites. Changing > causes this statement to return true when it should be false and vice versa.

ROR_11 to ROR_15 correspond to the following part of the code:

```
if (this.getDay() < 0 || this.getDay() [redacted] 31) {
```

All these mutants are killed by all test suites. The reason is that all test suites contain a test case where the value of day is equal to 31. Therefore this is tested by all, and killed.

ROR_16 to ROR_20 correspond to the following part of the code:

```
if (this.getDay() [redacted] 30) {
                        this.setDay( 1 );
                        this.setMonth( this.getMonth() + 1 );
                    }
```

There are no test cases in the Base Choice test suite that contain a 30 day month and the day is 30. Therefore these mutants are not killed by the Base Choice test suite.

ROR_17 changes the line from ==30 to =>30, since it is not possible to have a day that is greater than 31 (this is previously checked), this change does not have any impact. Therefore this mutant is not killed by any of the test suites. ROR_19 is not killed because of the same reason but this time the value ==30 is changed to <=30. ROR_17 and ROR_19 are equivalent.

The rest of the mutants in this category are killed by the Each Choice and Pair Wise test suites. This is because they contain a test case that covers this part of the code and changing the equals operator to anything but the above mentioned operators, changes the functionality.

ROR_21 to ROR_25 correspond to the following part of the code:

```
if (this.getYear() [redacted] 2100) {
```

There are no test cases that have the lead to the next date's year equal to 2100, therefore the mutants that involve changing the == to => and > are not killed by any test suite. However, changing == to < will make this program to return true when it is in fact false. Mutants ROR_21, ROR_22 are not killed by any test suite, and ROR_23 to ROR_25 are killed by all test suites except the Base Choice test suite as a result of the above mentioned explanation.

These mutants are not killed by the Base Choice test suite since it does not contain a test case where the month is December and date is equal to 31.

ROR_26 to ROR_30 correspond to the following part of the code:

```
else {
                    if (this.getDay() [redacted] 28) {
```

These test cases are not killed by the Base Choice test suite because it does not contain any test cases that is February and day equal to 28.

ROR_29 is not killed by any of the test suites. The program checks the *if* statement (before else) and since it is not possible for the day to be less than 28 (since the if statement is not true) this mutant does not make a difference in functionality making it an equivalent mutant. Therefore it is not killed by any of the test suites.

ROR_27 is only killed by the Pair Wise test suite because this test suite contains a test case where the value of date is greater than 28 and is February and is leap year. The Each Choice test suite does not have such a test case.

The rest of the mutants in this category are killed by the Each Choice and Pair Wise test suites.

ROR_31 corresponds to the following lines of code: `if (this.isLeapYear() == true) {`

Since there are no test cases where the month is February, day is 28 and is a not a leap year in the Base Choice test suite, this mutant is not killed. This mutant is killed by the Each Choice and Pair Wise test suites.

ROR_32 to ROR_36 correspond to the following part of the code:

```
else {
if (this.getDay() <= 29) {
```

There are no test cases where the month is February, day is 29 and is a leap year in either the Base Choice or Each Choice test suites. These mutants are therefore not killed by these two test suites.

ROR_33 is not killed by the Pair Wise test suite because there are no cases where the month is February, is leap year and the date is greater than 29.

ROR_35 is not killed because of the same reason, except date less than 29 and is not 28.

The rest of the mutants in this category are killed by the Pair Wise test suite since it contains a test case for February, leap year and day is equal to 29.

ROR_37 correspond to the following part of the code:

```
if (this.getDay() == 29) {
        if (this.isLeapYear() [redacted] true) {
```

This mutant is not killed by any of the test suites but the Pair Wise test suite. This is the same reason as ROR_35.

ROR_38 to ROR_42 correspond to the following part of the code: `if (this.getDay() [redacted] 29)`

These mutants are not killed by any of the test suites because they have no test cases that reach this point of the code.

COR_1 to COR_2 correspond to the following part of the code:

```
if (this.getYear() < 1508 [redacted] this.getYear() > 2100) {
```

COR_3 to COR_4 correspond to the following part of the code:

```
if (this.getMonth() < 1 [redacted] this.getMonth() > 12) {
```

COR_5 and COR_6 correspond to the following part of the code:

```
if (this.getDay() < 0 [redacted] this.getDay() > 31) {
```

Since COR_1 to COR_6 involve changing the || to && and ^ sign, we will look at them as one category.

Since changing || to && changes the logic, these mutants are caught by all test suites. It is not possible for both conditions to be true. These include COR_1, COR_3 and COR_5.

Changing || to ^ does not make any difference to the logic in this case. This is because ^ is an exclusive or operator, which in our case is true. We cannot have both values returning true. Therefore Mutants COR_2, COR_4 and COR_6 are not killed by any of the test suites because they are equivalent.

COI_1 to COI_3 correspond to the following part of the code:

```
if (this.getYear() < 1508 || this.getYear() > 2100) {
```

COI_4 to COI_6 correspond to the following part of the code:

```
if (this.getMonth() < 1 || this.getMonth() > 12) {
```

COI_7 to COI_9 correspond to the following part of the code:

```
if (this.getDay() < 0 || this.getDay() > 31) {
```

COI_10 correspond to the following part of the code:

```
case 10 :
                if (this.getDay() < 31) {
```

COI_11 correspond to the following part of the code:

```
case 11 :
                if (this.getDay() < 30) {
```

COI_12 correspond to the following part of the code:

```
else {
                if (this.getDay() == 30) {
```

COI_13 correspond to the following part of the code:

```
case 12 :
                if (this.getDay() < 31) {
```

COI_14 correspond to the following part of the code:

```
if (this.getYear() == 2100) {
```

COI_15 correspond to the following part of the code:

```
case 2 :
                if (this.getDay() < 28) {
```

COI_16 correspond to the following part of the code:

```
else {
                if (this.getDay() == 28) {
```

COI_17 correspond to the following part of the code: `if (this.isLeapYear() == true) {`

COI_18 correspond to the following part of the code: `if (this.getDay() == 29) {`

COI_19 correspond to the following part of the code: `if (this.isLeapYear() == true)`

COI_20 correspond to the following part of the code:

```
else {
if (this.getDay() > 29) {
```

COI_1 to COI_20 involve adding the ! operator, therefore we will talk about them all at the same time. Since adding this operator changes the logic of statements, it should be easy to catch all mutants. However, some test suites do not have test cases covering some parts of the code. Where there is no code coverage (please look at the mutants for ROR, the parts of the code that are not used by each test suite are identified), the mutants are not killed by the test suite. If the code is covered by a test case, the mutants are killed. COI_1 to COI_10, COI_11 and COI_13 are killed by all test suites.

COI_14, 16, 17, 18, 19 and 20 are not killed by the Base Choice test suite. COI_18, 19 and 20 are not killed by the Each Choice test suite. Lastly COI_20 is not killed by any of the test suites.

COI_20 is not killed because this case is an error case. The combination of February and 29 is not feasible.

APPENDIX D PACK HEX CHAR CASE STUDY

D1.CP Specification

Parameter	Categories	Choices	Constraints
Input string	[Cat1]Length	[ch1.1]Even	[evenLength][Base Choice]
		[ch1.2]Odd	[oddLength]
		[ch1.3]Zero	[single]
	[Cat2]Type	[ch2.1]All hexadecimal	[Base Choice]
		[ch2.2]Mix of hexadecimal and non hexadecimal	[single]
		[ch2.3]No hexadecimal numbers	[error]
	[Cat3]Case	[ch3.1]All capital	[Base Choice]
		[ch3.2]All lower case	
		[ch3.3]Mix of capital and lowercase	
RLEN	[Cat4]Length	[ch4.1]Zero	[single]
		[ch4.2]Less than string length	[Base Choice]
		[ch4.3]Equal to string length	
		[ch4.4]Invalid (negative, greater than string length)	[invalidRLEN][error]
ODD_DIGIT	[Cat5]value	[ch5.1]Hexadecimal value	[Base Choice]
		[ch5.2]-1	
		[ch]Invalid (less than -1, not hexadecimal)	[invlaidODD_DIGIT][error]

Table 52 PackHexChar Case Study: CP Specification

D2.Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.1 |C 3.1 |C 4.2 |C 5.1 |	3, 14, "4ABC"	[228, 171] -1
2	C 1.2 |C 2.1 |C 3.1 |C 4.2 |C 5.1 |	4, 14, "4ABD4"	[228, 171] 13
3	C 1.1 |C 2.1 |C 3.2 |C 4.2 |C 5.1 |	3, 14, "4abc"	[228, 171] -1
4	C 1.1 |C 2.1 |C 3.3 |C 4.2 |C 5.1 |	3, 14, "4abC"	[228, 171] -1
5	C 1.1 |C 2.1 |C 3.1 |C 4.1 |C 5.1 |	0, 14, "4ABC"	[] 14
6	C 1.1 |C 2.1 |C 3.1 |C 4.3 |C 5.1 |	4, 14, "4ABC"	[228, 171] 12
7	C 1.1 |C 2.1 |C 3.1 |C 4.2 |C 5.2 |	3, -1, "4ABC"	[74] 11
8	C 1.1 |C 2.2 |C 3.1 |C 4.2 |C 5.1 |	0, -1, "4xBC"	[] -3
9	C 2.3 |	2, 1, "hi!"	[] -3
10	C 4.4 |	-1, 1, "4ABC"	[] -2
11	C 5.3 |	2, -2, "4ABC"	[]

			-3

Table 53 PackHexChar Case Study: Base Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.1 |C 3.2 |C 4.3 |C 5.2 |	4, -1, "3abf"	[58, 191] -1
2	C 1.2 |C 2.1 |C 3.3 |C 4.1 |C 5.1 |	0, 12, "3Ab"	[] 12
3	C 1.2 |C 2.1 |C 3.1 |C 4.2 |C 5.2 |	3, -1, "3ABF"	[58] 11
4	C 1.2 |C 2.2 |C 3.2 |C 4.3 |C 5.2 |	3, -1, "3ab"	[58] 11
5	C 2.3 |	2, 1, "hi!"	[] -3
6	C 4.4 |	-1, 1, "4ABC"	[] -2
7	C 5.3 |	2, -2, "4ABC"	[] -3

Table 54 PackHexChar Case Study: Each Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.1 |C 3.2 |C 4.3 |C 5.2 |	5, -1, "3ab2f"	[58, 178] 15
2	C 1.2 |C 2.1 |C 3.1 |C 4.2 |C 5.2 |	4, -1, "3AB2F"	[58, 178]-1
3	C 1.1 |C 2.1 |C 3.1 |C 4.3 |C 5.1 |	4, 2, "3AB2"	[35, 171]2
4	C 1.1 |C 2.1 |C 3.3 |C 4.2 |C 5.1 |	3, 11, "3Ab2"	[179, 171] -1
5	C 1.1 |C 2.1 |C 3.1 |C 4.1 |C 5.2 |	0, -1, "3AB2"	[] -1
6	C 1.1 |C 2.1 |C 3.2 |C 4.1 |C 5.1 |	0, 1, "fabdea"	[] 1
7	C 1.2 |C 2.1 |C 3.3 |C 4.3 |C 5.1 |	0, 1, "fabdea"	[218] 11
8	C 1.2 |C 2.1 |C 3.3 |C 4.1 |C 5.2 |	2, 13, "abce"	[218] 11
9	C 1.1 |C 2.1 |C 3.2 |C 4.2 |C 5.1 |	2, 13, "abce"	[239, 163, 189] -1
10	C 1.1 |C 2.2 |C 3.3 |C 4.3 |C 5.1 |	5, 14, "fa3bde"	[] -3
11	C 2.3 |	4, 5, "9bAx"	[] -3
12	C 4.4 |	2, 1, "hi"	[] -2
13	C 5.3 |	-1, 10, "ab2d"	[] -3

Table 55 PackHexChar Case Study: Pair Wise Test Frames

D3.Explanation of Mutants Killed or Alive

There is one class level mutant generated for the next date problem which is IPC_1. Removing the `super( year, month, day )`, statement from the constructor means that the default values of year, month and day are set to zero. This means that the next date is zero as well, making the actual value equal to the test value. Therefore this mutant is not killed by any of the test suites.

AOIS_1 to AOIS_6 involve the following lines of code:

```
public NextDate( int year, int month, int day )
{
super( year, month, day );
}
```

Since these mutants make changes to the constructor, the same changes apply to the object that is created in the test case to compare against. This is why these mutants are not killed. These mutants are equivalent.

LOI_1 to LOI_3 and AOIU_1 to AOIU_3 are killed by all test suites, because the value of the parameters are negated and when there is an invalid value (-1986) the value returned is 0,0,0. When the value is compared to the correct value (even though this is the wrong value), the values are not the same. Therefore these mutants are killed.

AORB_1 to AORB_4 correspond to the following part of the code:

```
case 10:
if (this.getDay() < 31) {
                    this.setDay( this.getDay() [redacted] 1 );
                }
```

Since all test suites have a test case where the value of day is less than 31 and changing the addition returns a wrong value for day parameter, these mutants are killed by all test suites.

AORB_5 to AORB_8 correspond to the following part of the code:

```
else {
                    this.setDay( 1 );
                    this.setMonth( this.getMonth() + 1 );
                }
```

Since the Each Choice test suite does not contain any test cases where the month is a 31 day month and the day is 31, these mutants are not killed by this test suite. However these mutants are killed by the Base Choice and Pair wise test suites.

AORB_9 to AORB_12 correspond to the following part of the code:

```
case 11 :
if (this.getDay() < 30) {
  this.setDay( this.getDay() + 1 );
}
```

There are no test cases in test suite that reach this part of the code (i.e. no test cases that the month has 30 days and the day is less than 30). Therefore these mutants are not killed by this test suite. Since the Base Choice and Pair Wise test suites contain a test case with the above mentioned criterion, they are able to kill these mutants.

AORB_13 to AORB_16 correspond to the following part of the code:

```
else {
                    if (this.getDay() == 30) {
                        this.setDay( 1 );
                        this.setMonth( this.getMonth() + 1 );
                    }
```

No test cases correspond to this part of the code in the Base Choice test suite. Therefore these mutants are not killed. However the Each Choice and Pair Wise test suites contain a test case which covers this portion, therefore these mutants are killed by it.

AORB_17 to AORB_AORB_20 correspond to the following part of the code:

```
 case 12 :
                if (this.getDay() < 31) {
                    this.setDay( this.getDay() + 1 );
                }
```

These mutants are killed by the Pair Wise and Base Choice test suites, since they contain a test case that reaches this part of the code. The Each Choice test suite, however, does not contain such test case and is therefore not able to kill these mutants.

AORB_21 to AORB_24 correspond to the following part of the code:

```
else {
     this.setYear( this.getYear() + 1 );
                    }
```

There are no test cases in the Base Choice test suites that cover this part of the code. (i.e. no test cases covering the case where the month is equal to 12 and the day is not less than 31. Therefore these mutants are not caught by the Base Choice test suite. The Each Choice and Pair Wise test suites however contain a choice with these options. Therefore they are killed.

AORB_25 to AORB_28 correspond to the following part of the code:

```
case 2 :
```

```
if (this.getDay() < 28) {
this.setDay( this.getDay() + 1 );
                }
```

There are no test cases involving this part of the code in the Each Choice and Pair Wise test suites, therefore these mutants are not killed by these test suites. The Base Choice test suite however, kills all these mutants. By adding a choice to the CP specification where day is equal to 28, these mutants can be killed by the Pair Wise test suite. The Each Choice test suite however may not kill these mutants since it is not guaranteed that these two choices will appear in the same test frame (i.e. February 28).

AORB_29 to AORB_32 correspond to the following part of the code:

```
if (this.getDay() == 28) {
                        if (this.isLeapYear() == true) {
                            this.setDay( this.getDay() + 1 );
                        }
```

No test cases in the any of the test suites cover this part of the code. Therefore none of the test suites kill these mutants.

ROR_1 to ROR_5 correspond to the following part of the code:

```
if (this.getYear() < 1508 || this.getYear() >= 2100) {
```

There are no test cases in any of the test suite that have the value of year equal to 2100. Since ROR_1 changes the operator in the second statement from >= to >, this has no impact in the test cases and this mutant is not killed by any of the test suites. To kill this mutant the error choice can be changed to the year 2100.

ROR_2 is killed easily by all test suites, since the operator >= in the second statement is changed to < and this makes the statement return true where it should be false.

ROR_3 is killed by all test suites as well, because of the same reason as ROR_2, except this time the value of >= is changed to <=, which does not make a difference in this case.

ROR_4 and ROR_5 change >= to == and !=. These mutants are killed by all test suites because they all contain an error choice of year greater than 2100 and year not equal to 2100 and not valid.

ROR_6 to ROR_10 correspond to the following part of the code:

```
if (this.getMonth() < 1 || this.getMonth() > 12) {
```

```
System.out.println( "Invalid month" );}
```

Since all test suites contain a test case with value of month equal to 12 and less than 12, these mutants are killed by all test suites. Changing > causes this statement to return true when it should be false and vice versa.

ROR_11 to ROR_15 correspond to the following part of the code:

```
if (this.getDay() < 0 || this.getDay() > 31) {
```

All these mutants are killed by all test suites. The reason is that all test suites contain a test case where the value of day is equal to 31. Therefore this is tested by all, and killed.

ROR_16 to ROR_20 correspond to the following part of the code:

```
if (this.getDay() == 30) {
                        this.setDay( 1 );
                        this.setMonth( this.getMonth() + 1 );
                    }
```

There are no test cases in the Base Choice test suite that contain a 30 day month and the day is 30. Therefore these mutants are not killed by the Base Choice test suite.

ROR_17 changes the line from ==30 to =>30, since it is not possible to have a day that is greater than 31 (this is previously checked), this change does not have any impact. Therefore this mutant is not killed by any of the test suites. ROR_19 is not killed because of the same reason but this time the value ==30 is changed to <=30. ROR_17 and ROR_19 are equivalent.

The rest of the mutants in this category are killed by the Each Choice and Pair Wise test suites. This is because they contain a test case that covers this part of the code and changing the equals operator to anything but the above mentioned operators, changes the functionality.

ROR_21 to ROR_25 correspond to the following part of the code:

```
if (this.getYear() == 2100) {
```

There are no test cases that have the lead to the next date's year equal to 2100, therefore the mutants that involve changing the == to => and > are not killed by any test suite. However, changing == to < will make this program to return true when it is in fact false. Mutants ROR_21, ROR_22 are not killed by any test suite, and ROR_23 to ROR_25 are killed by all test suites except the Base Choice test suite as a result of the above mentioned explanation.

These mutants are not killed by the Base Choice test suite since it does not contain a test case where the month is December and date is equal to 31.

ROR_26 to ROR_30 correspond to the following part of the code:

```
else {
                    if (this.getDay() [redacted] 28) {
```

These test cases are not killed by the Base Choice test suite because it does not contain any test cases that is February and day equal to 28.

ROR_29 is not killed by any of the test suites. The program checks the *if* statement (before else) and since it is not possible for the day to be less than 28 (since the if statement is not true) this mutant does not make a difference in functionality making it an equivalent mutant. Therefore it is not killed by any of the test suites.

ROR_27 is only killed by the Pair Wise test suite because this test suite contains a test case where the value of date is greater than 28 and is February and is leap year. The Each Choice test suite does not have such a test case.

The rest of the mutants in this category are killed by the Each Choice and Pair Wise test suites.

ROR_31 corresponds to the following lines of code: `if (this.isLeapYear() [redacted] true) {`

Since there are no test cases where the month is February, day is 28 and is a not a leap year in the Base Choice test suite, this mutant is not killed. This mutant is killed by the Each Choice and Pair Wise test suites.

ROR_32 to ROR_36 correspond to the following part of the code:

```
else {
if (this.getDay() [redacted] 29) {
```

There are no test cases where the month is February, day is 29 and is a leap year in either the Base Choice or Each Choice test suites. These mutants are therefore not killed by these two test suites.

ROR_33 is not killed by the Pair Wise test suite because there are no cases where the month is February, is leap year and the date is greater than 29.

ROR_35 is not killed because of the same reason, except date less than 29 and is not 28.

The rest of the mutants in this category are killed by the Pair Wise test suite since it contains a test case for February, leap year and day is equal to 29.

ROR_37 correspond to the following part of the code:

```
if (this.getDay() == 29) {
        if (this.isLeapYear() [illegible] true) {
```

This mutant is not killed by any of the test suites but the Pair Wise test suite. This is the same reason as ROR_35.

ROR_38 to ROR_42 correspond to the following part of the code:

```
if (this.getDay() [illegible] 29) {
```

These mutants are not killed by any of the test suites because they have no test cases that reach this point of the code.

COR_1 to COR_2 correspond to the following part of the code:

```
if (this.getYear() < 1508 || this.getYear() > 2100) {
```

COR_3 to COR_4 correspond to the following part of the code:

```
if (this.getMonth() < 1 || this.getMonth() > 12) {
```

COR_5 and COR_6 correspond to the following part of the code:

```
if (this.getDay() < 0 || this.getDay() > 31) {
```

Since COR_1 to COR_6 involve changing the || to && and ^ sign, we will look at them as one category.

Since changing || to && changes the logic, these mutants are caught by all test suites. It is not possible for both conditions to be true. These include COR_1, COR_3 and COR_5.

Changing || to ^ does not make any difference to the logic in this case. This is because ^ is an exclusive or operator, which in our case is true. We cannot have both values returning true. Therefore Mutants COR_2, COR_4 and COR_6 are not killed by any of the test suites because they are equivalent.

COI_1 to COI_3 correspond to the following part of the code:

```
if (this.getYear() < 1508 || this.getYear() > 2100) {
```

COI_4 to COI_6 correspond to the following part of the code:

```
if (this.getMonth() < 1 || this.getMonth() > 12) {
```

COI_7 to COI_9 correspond to the following part of the code:

```
if (this.getDay() < 0 || this.getDay() > 31) {
```

COI_10 correspond to the following part of the code:

```
case 10 :
if (this.getDay() < 31) {
```

COI_11 correspond to the following part of the code:

```
case 11 :
if (this.getDay() < 30) {
```

COI_12 correspond to the following part of the code:

```
else {
if (this.getDay() == 30) {
```

COI_13 correspond to the following part of the code:

```
case 12 :
if (this.getDay() < 31) {
```

COI_14 correspond to the following part of the code: `if (this.getYear() == 2100) {`

COI_15 correspond to the following part of the code:

```
case 2 :
if (this.getDay() < 28) {
```

COI_16 correspond to the following part of the code:

```
else {
if (this.getDay() == 28) {
```

COI_17 correspond to the following part of the code: `if (this.isLeapYear() == true) {`

COI_18 correspond to the following part of the code: `if (this.getDay() == 29) {`

COI_19 correspond to the following part of the code: `if (this.isLeapYear() == true) {`

COI_20 correspond to the following part of the code:

```
else {
if (this.getDay() > 29) {
```

COI_1 to COI_20 involve adding the ! operator, therefore we will talk about them all at the same time. Since adding this operator changes the logic of statements, it should be easy to catch all mutants. However, some test suites do not have test cases covering some parts of the code. Where there is no code coverage (please look at the mutants for ROR, the parts of the code that are not used by each test suite are identified), the mutants are not killed by the test suite. If the code is covered by a test case, the mutants are killed. COI_1 to COI_10, COI_11 and COI_13 are killed by all test suites.

COI_14, 16, 17, 18, 19 and 20 are not killed by the Base Choice test suite. COI_18, 19 and 20 are not killed by the Each Choice test suite. Lastly COI_20 is not killed by any of the test suites.

COI_20 is not killed because this case is an error case. The combination of February and 29 is not feasible.

APPENDIX E ORDSET CASE STUDY

E1. CP Specification

Parameters	Categories	Choices	Properties	Selectors
S1	[cat1] Number of Elements	[Ch1.1] zero	[firstSetEmpty]	
		[Ch1.2] average	[firstSetAverage]	[Base Choice]
		[Ch1.3] large	[firstSetLarge] [single]	
S2	[cat2] Number of Elements	[Ch2.1] zero	[secondSetEmpty]	
		[Ch2.2] average	[secondSetAverage]	[Base Choice]
		[Ch2.3] large		[secondSetLarge] [single]
S3	[cat3] Common elements	[Ch3.1] S1 has elements not in S2		[Base Choice][if !firstSetEmpty]
		[Ch3.2] S2 has elements not in S1		[if !secondSetEmpty]
		[Ch3.3] s1 and s2 have no elements in common		[if !firstSetEmpty && !secondSetEmpty] [single]
		[Ch3.4] Sets have all elements in common		[if (firstSetEmpty && secondSetEmpty)||(firstSetAverage && secondSetAverage)||(firstSetLarge&&secondSetLarge)]

Table 56 OrdSet Case Study: CP Specification

E2. Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.2 |C 3.1 |	{ 3, 4, 19, 44, 84 } { 1, 4, 12, 19, 44, 80, 90, 125, 284, 420 }	{ 3, 84 }
2	C 1.2 |C 2.1 |C 3.1 |	{ 1, 4 } { 1, 3, 9, 18 }	{ 4 }
3	C 1.2 |C 2.2 |C 3.2 |	{ 1, 2, 3, 4 } { 1, 2 }	{ 3, 4 }
4	C 1.2 |C 2.2 |C 3.3 |	{ 3, 9 } {}	{ 3, 9 }
5	C 1.3 |C 2.2 |C 3.1 |	{ 12, 15, 23 }; { 19, 57, 86 };	{ 12, 15, 23 }
6	C 1.2 |C 2.3 |C 3.1 |	{ 1, 4, 8, 23, 47, 52, 72, 85, 91,203 } { 1, 3, 9, 10 }	{ 4, 8, 23, 47, 52, 72, 85, 91, 203 }
7	C 1.2 |C 2.2 |C 3.4 |	{ 1, 3, 4, 9 } { 1, 3, 4, 9 }	{}

Table 57 OrdSet Case Study: Base Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.2| C 3.3 |	{} {3,4,9,11}	{}
2	C 1.2 |C 2.2| C 3.2 |	{1,3,4,18,23} {1,3,4,18,23,33}	{}

3	C 1.2 |C 2.1| C 3.1 |	{1,3,4} {}	{1,3,4}
4	C 1.1 |C 2.1| C 3.4 |	{} {}	{}
5	C 1.3 |C 2.3| C 3.1 |	{42,17,472,751,413,12,68,81,38,40} {42,17,23,13,16,73,63,91,28,34}	{472,751,413,12,68 ,81,38,40}

Table 58 OrdSet Case Study: Each Choice Test Frames

#	Test frame	Test Case	Output
1	C 1.1 |C 2.1 |C 3.4 |	{} {}	{}
2	C 1.2 |C 2.2 |C 3.4 |	{1,3,4,9} {1,3,4,9}	{}
3	C 1.2 |C 2.1 |C 3.1 |	{25,84,99} {}	{25,84,99}
4	C 1.1 |C 2.2 |C 3.2 |	{} {41,50}	{}
5	C 1.2 |C 2.2 |C 3.2 |	{4,5} {4,5,9}	{}
6	C 1.2 |C 2.2 |C 3.1 |	{2,3,9} {2,3}	{9}
7	C 1.3 |C 2.1 |C 3.1 |	{1,11,23,45,57,68,79,81,99,101,104} {}	{1,11,23,45,57,68, 79,81,99,101,104}
8	C 1.3 |C 2.3 |C 3.3 |	{1,11,23,45,57,68,79,81,99,101,104} {1,11,23,45,57,68,79,81,99,101,104}	{}

Table 59 OrdSet Case Study: Pair Wise Test Frames

E3. Explanation of Mutants Killed or Alive

IOD_1 removes the `toString` method from the class. Since this method is used for comparison, removing this method will cause the system to use the built-in `toString` which will return an invalid string. Therefore this mutant is killed by all test suites.

IOD_2 removes the `equals` method from the class. Since this method is never used, this mutant is not killed by any of the test suites.

JSI_1 to JSI_3 change variables from `int` value to `static int`.

Since `_set_size` and `_last` are not changed anywhere in the program, changing the value to static has no impact on the variables. Since this change does not have any impact on the functionality of the program, these mutants are not killed by any of the test suites. JSI_1 and JSI_3 are therefore equivalent.

The array _set however is used later in the code and changing this value to static does not allow further change to the variable. Therefore JSI_2 is killed by all test suites.

```
AODU_4:90:int_binSearch(int):-1 => 1
```

```
AODU_5:103:int_binSearch(int):-1 => 1
```

AODU_4 and AODU_5 are killed by all test suites. All test suites reach this part of the code because there is a test case where both sets are non-empty and the resulting difference set is also non-empty. Therefore all test suites are able to kill these mutants.

```
AORB_1:40:int_make_a_free_slot(int):j + 1 => j * 1
AORB_2:40:int_make_a_free_slot(int):j + 1 => j / 1
AORB_3:40:int_make_a_free_slot(int):j + 1 => j % 1
AORB_4:40:int_make_a_free_slot(int):j + 1 => j - 1
```

These mutants are not killed by the Base Choice and Pair Wise test suites but are killed by the Each Choice test suite.

The values of inputs where chosen so that they are ordered, except for one test case which resides in the Each Choice test suite. If the set is ordered we will never reach this part of the code and this is why these mutants are only killed by the Each Choice test suite. Changing + to other operators will cause the statement to set the wrong value. Therefore these mutants are killed by the Each Choice test suite.

```
AORB_5:44:int_make_a_free_slot(int):_last + 1 => _last * 1
AORB_6:44:int_make_a_free_slot(int):_last + 1 => _last / 1
AORB_7:44:int_make_a_free_slot(int):_last + 1 => _last % 1
AORB_8:44:int_make_a_free_slot(int):_last + 1 => _last - 1
```

These are killed by all test suites. Since all test suites contain a choice that reaches this part of the code and changing the value of last has an impact on the ending result, these mutants are killed.

```
AORB_9:57:void_addElement(int):_last + 1 => _last * 1
AORB_10:57:void_addElement(int):_last + 1 => _last / 1
AORB_11:57:void_addElement(int):_last + 1 => _last % 1
AORB_12:57:void_addElement(int):_last + 1 => _last - 1
```

These mutants are not killed by any of the test suites. Since the test cases are not targeted at the `addElement` method, there are no test cases where the size of the array is set to a value that is larger than the elements being added to the array. This is why this part of the code is never reached and therefore not killed by any of the test suites.

```
AORB_13:72:int_getSize():_last + 1 => _last * 1
AORB_14:72:int_getSize():_last + 1 => _last / 1
AORB_15:72:int_getSize():_last + 1 => _last % 1
AORB_16:72:int_getSize():_last + 1 => _last - 1
```

Changing the size of the array will have an impact on the final difference set's size. Since all test suites contain a test case where the number of elements in the difference set is greater than 0, these mutants are killed by all test suites.

```
AORB_17:93:int_binSearch(int):i + j => i * j
AORB_18:93:int_binSearch(int):i + j => i / j
AORB_19:93:int_binSearch(int):i + j => i % j
AORB_20:93:int_binSearch(int):i + j => i - j
AORB_21:93:int_binSearch(int):(i + j) / 2 => (i + j) * 2
AORB_22:93:int_binSearch(int):(i + j) / 2 => (i + j) % 2
AORB_23:93:int_binSearch(int):(i + j) / 2 => i + j + 2
AORB_24:93:int_binSearch(int):(i + j) / 2 => i + j - 2
```

Since these mutants change the binary search algorithm and meddle with the boundary size of the search, the value being searched for will not be found and the program may end up in an infinite loop. This mutant is therefore killed by all test suites.

```
AORB_25:95:int_binSearch(int):m + 1 => m * 1
AORB_26:95:int_binSearch(int):m + 1 => m / 1
AORB_27:95:int_binSearch(int):m + 1 => m % 1
AORB_28:95:int_binSearch(int):m + 1 => m - 1
```

These mutants are killed by all test suites. This is because of the same reason as AORB_17 to AORB_24.

```
AORS_1:18:OrdSetSimple(int):k++ => k-
```

Decrementing k instead of incrementing causes the loop to be infinite since k will never be greater than the set's size. Therefore this mutant is killed by all test suites.

```
AORS_2:33:int_make_a_free_slot(int):i++ => i-
```

Since all test suites have a test case with non-empty sets, and changing i++ to i—will cause the program to go into infinite loop, this mutant is killed by all test suites.

```
AORS_3:39:int_make_a_free_slot(int):j-- => j++
```

This is not killed by the Base Choice and Pair Wise test suite. They are killed by the Each Choice test suite.

Since the Each Choice test suite has a test case where the values are not ordered, this part of the code is reached by this test suite. Changing the j- - to j++ causes the system to go in an infinite loop and therefore is killed by this test suite.

The Base Choice and Pair Wise test suites however do not contain such a test suite and therefore are not able to kill this mutant.

```
AORS_4:46:int_make_a_free_slot(int):_last++ => _last-
AORS_5:114:OrdSetSimple_difference(OrdSetSimple):k++ => k-
AORS_6:126:java.lang.String_toString():k++ => k-
AORS_7:148:boolean_equals(java.lang.Object):i++ => i-
```

AORS_4 to AORS_6 are killed by all tests because of the same reason as AORS_1 and AORS_2. AORS_7 is not killed by any of the test suites because the `equals` method is never used.

```
AODU_1:21:OrdSetSimple(int):-1 => 1
```

Changing -1 to 1 for variable `_last` will cause the last two items to not be added to the set. Since all test suites have test cases where the set difference value is non-empty, this mutant is killed by all test suites.

```
AODU_2:31: int_getElementAt(int):-1 => 1
```

This mutant is not killed by any of the test suites. This is because there are no test cases where a value of less than or greater than the size is requested.

```
AODU_3:90:int_binSearch(int):-1 => 1
```

This mutant is killed by the Base Choice and Pair Wise test suites because they have a test case where the difference set only has one value. This causes the binary search to look for an item that is out of bound, i.e. 1.

```
AOIU_1:16:OrdSetSimple(int):size => -size
AOIU_2:17:OrdSetSimple(int):_set_size => -_set_size
```

AOIU_1 and AOIU_2 change the size of the set to a negative value which is invalid. Since all test suites use this value to set the size of the ordered set, these mutants are killed by all test suites.

```
AOIU_3:29:int_make_a_free_slot(int):val => -val
```

Since there are no duplicates, this statement will never return true for any of the values. Therefore this is not caught by any of the mutants.

```
AOIU_4:39:int_make_a_free_slot(int):_last => -_last
```

Changing the value of last to negative will cause the program to access a negative value for an array which is not possible. This is why this mutant is killed by all test suites.

```
AOIU_5:40:int_make_a_free_slot(int):j => -j
AOIU_6:42:int_make_a_free_slot(int):i => -i
```

Since there are no test cases in the Base Choice and Pair Wise test suites where the set values of the first set are unordered, these mutants are not killed by any of the test suites. This is because without this condition the program will never reach this part of the code.

```
AOIU_7:44:int_make_a_free_slot(int):_last => -_last
AOIU_8:47:int_make_a_free_slot(int):pos => -pos
```

```
AOIU_9:62:void_addElement(int):n => -n
AOIU_10:64:void_addElement(int):n => -n
AOIU_11:72:int_getSize():_last => -_last
AOIU_12:80:int_getElementAt(int):i => -i
AOIU_13:87:int_binSearch(int):_last => -_last
AOIU_14:93:int_binSearch(int):i => -i
AOIU_15:95:int_binSearch(int):m => -m
AOIU_16:97:int_binSearch(int):m => -m
AOIU_17:101:int_binSearch(int):i => -i
AOIU_18:112:OrdSetSimple_difference(OrdSetSimple):size1 => -size1
AOIU_19:116:OrdSetSimple_difference(OrdSetSimple):k => -k
```

All test suites have a test case that has a non-empty difference set. Changing the values of variables in the mutants above will have an impact on the number of values in the set difference ordered set and are therefore killed by all test suites.

```
AOIS_1:16:OrdSetSimple(int):size => ++size
```

Increasing the size will have no effect on the final size of the difference set. Therefore this mutant is not killed by any test suite.

```
AOIS_2:16:OrdSetSimple(int):size => --size
```

Decreasing the size of the array will cause the last element to be disregarded. Since each test suite has a test case where the last element is contained in the difference set, this mutant is killed by all test suites.

```
AOIS_3:16:OrdSetSimple(int):size => size++
AOIS_4:16:OrdSetSimple(int):size => size-
```

AOIS_3 and AOIS_4 are not killed by any of the test suites because of the reason mentioned for AOIS_1.

```
AOIS_5:17:OrdSetSimple(int):_set_size => ++_set_size
AOIS_6:17:OrdSetSimple(int):_set_size => --_set_size
AOIS_7:17:OrdSetSimple(int):_set_size => _set_size++
AOIS_8:17:OrdSetSimple(int):_set_size => _set_size-
```

Increasing the size of the set will have no impact on the final value of the difference set this is why AOIS_5 is not killed by any of the test suites.

AOIS_6 to AOIS_8 are killed by all test suites.

Decreasing the size will cause the last value to be missed in the difference set. This is why AOIS_6 is killed.

AOIS_7 will define a set with `set_size` and then increase the value. This causes the loop in the following line to try to access a value that does not exist. Therefore this mutant is killed by all test suites.

AOIS_8 will define a set with `set_size` and then decrease the value. This causes one item to be missed when setting the values for the set. Therefore this mutant is killed by all test suites.

```
AOIS_9:18:OrdSetSimple(int):k => ++k
AOIS_10:18:OrdSetSimple(int):k => --k
AOIS_11:18:OrdSetSimple(int):k => k++
AOIS_12:18:OrdSetSimple(int):k => k--
AOIS_13:18:OrdSetSimple(int):_set_size => ++_set_size
AOIS_14:18:OrdSetSimple(int):_set_size => --_set_size
AOIS_15:18:OrdSetSimple(int):_set_size => _set_size++
AOIS_16:18:OrdSetSimple(int):_set_size => _set_size-
```

AOIS_9 is not killed by any of the test suites. If there was a test case where both sets contained only one value and the value in the first set did not appear in the second set, this mutant would be killed. In this case the program tries to access the set at 1, which does not exist resulting in array out of bound exception.

AOIS_10 to AOIS_12 are killed by all test suites. AOIS_10 causes the system to try and access the set at -1 location and causes an error and is therefore killed by all test suites. AOIS_11 and AOIS_12 add a shortcut operator after the variable. Adding a ++ operator will cause the value of k to increase by two and – operator causes the value of k to stay the same and therefore an infinite loop. Therefore these two mutants are killed by all test suites.

This is the same case for AOIS_13 to AOIS_16 and these mutants are therefore killed by all test suites.

```
AOIS_17:29:int_make_a_free_slot(int):val => ++val
AOIS_18:29:int_make_a_free_slot(int):val => --val
AOIS_19:29:int_make_a_free_slot(int):val => val++
AOIS_20:29:int_make_a_free_slot(int):val => val-
```

AOIS_17 and AOIS_19 are not killed by the Base Choice and Pair Wise test suites but are killed by the Each Choice test suite. Since the Each Choice test suite contains a choice with unordered elements, these mutants are caught.

AOIS_18 is killed by the Base Choice and Each Choice test suite but not by the Pair Wise test suites. This is because the Base Choice and Each Choice test suite contain a test case where the values in the difference set are consecutive values. The program adds elements that do not exist

in the set. If there are consecutive values in the set, when the program is adding the elements, it looks for value-1 and since it exists in the set, it does not add the element to the set.This causes the values in the difference set to be invalid.

AOIS_20 is not killed by any of the test suites. If the program enters this break as a result of the mutant, the program will go to the if statement and continue normal execution without the value of `val` used. Therefore this mutant is equivalent and not killed by any of the test suites.

```
AOIS_21:30:int_make_a_free_slot(int):pos => ++pos
AOIS_22:30:int_make_a_free_slot(int):pos => --pos
AOIS_23:30:int_make_a_free_slot(int):pos => pos++
AOIS_24:30:int_make_a_free_slot(int):pos => pos-
```

Adding shortcut operators after the variable makes the change to the variable after the comparison is done. If the value of `pos` is greater or equal to 0 then it returns 1 and the `pos` value is never used again. If the `pos` value is less than 0 the program changes the value of `pos` to a value that is not dependent on the current value of pos. Therefore AOIS_23 and AOIS_24 are considered to be equivalent and are not killed by any of the test suites.

AOIS_21 is killed by all test suites. Since the value of position is increased by 1, the program tries to add item to the second position which does not exist. Therefore this mutant is killed.

AOIS_22 to AOIS_24 are not killed by any of the test suites because this part of the code is not covered.

```
AOIS_25:33:int_make_a_free_slot(int):i => ++i
AOIS_26:33:int_make_a_free_slot(int):i => --i
AOIS_27:33:int_make_a_free_slot(int):i => i++
AOIS_28:33:int_make_a_free_slot(int):i => i--
AOIS_29:33:int_make_a_free_slot(int):_last => ++_last
AOIS_30:33:int_make_a_free_slot(int):_last => --_last
AOIS_31:33:int_make_a_free_slot(int):_last => _last++
AOIS_32:33:int_make_a_free_slot(int):_last => _last-
AOIS_25 to AOIS_29 and AOIS_31 are killed by all test suites.
```

Changing the value of i will cause the loop to only execute once. Therefore AOIS_25 to AOIS_28 are killed by all test suites.

If the value of last is increased, i will never be less than _last and therefore there will be no spot for adding a new value. Therefore AOIS_29 and AOIS_31 are killed.

Decreasing last will cause no values to be added to any of the sets and therefore the difference set will also be empty. Therefore AOIS_30 and AOIS_32 are killed by all test suites.

```
AOIS_33:34:int_make_a_free_slot(int):i => ++i
AOIS_34:34:int_make_a_free_slot(int):i => --i
AOIS_35:34:int_make_a_free_slot(int):i => i++
AOIS_36:34:int_make_a_free_slot(int):i => i--
AOIS_37:34:int_make_a_free_slot(int):val => ++val
AOIS_38:34:int_make_a_free_slot(int):val => --val
AOIS_39:34:int_make_a_free_slot(int):val => val++
AOIS_40:34:int_make_a_free_slot(int):val => val-
```

AOIS_33 and AOIS_35 are killed by the Each Choice test suite but not by the Base Choice and Pair Wise test suites. This is because the Each Choice test suite contains an unordered set and the value of set at i+1 can be greater than the value.

AOIS_37 and AOIS_39 are killed by the Each Choice test suite but not by the Base Choice and Pair Wise test suites. These are killed because of the same reason as AOIS_33 and AOIS_35.

AOIS_34, AOIS_36, AOIS_38 are killed by all test suites.

Since all test suites contain ordered sets and the value of set at i-1 is always less than the value that is being added, these mutants are killed by all test suites.

AOIS_40 is killed by the Pair Wise test suite but not the Each Choice and Base Choice test suites.

This mutant resides in the `make_a_free_slot method`. When adding values to a set, this function makes sure the set does not already contain the value and finds a place for the new value so that the set is ordered. This particular part of the code identifies the index in the ordered set where to place the new element, i.e., the index of the set where all the elements with a lower index have a smaller value than this new element. This part of the code goes through a loop and compares the integer value to be added, referred to as newValue, to the integer values in the set at increasing indexes, referred to as existValue(i). Each time the loop iterates the mutant decreases newValue by one after each comparison: i.e., if we have 4 elements in the set, newValue is decreased 3 times by 1 before the 4^{th} comparison and the last comparison is therefore with newValue-3. This results in unordered sets under a specific condition which we identified to be the following: assuming we have to values V1 and V2, V2>V1 to be added in the set, and V1 being at index i in the set, then if V2-1>V1 the mutant produces an unordered set (value V2 will be inserted before value V1). Having unordered sets in memory is not enough to reveal the fault. The difference function looks at all the elements of the first set in sequence and

tries to find them in the second set with a binary search. Regardless of whether the first set is ordered or not in memory, this does not help revealing the fault. We have identified we need the second set to be unordered, i.e., to satisfy the condition above. This is however not sufficient. We also need the common part of the two sets to be in the unordered part of the second set. This is even not sufficient. In fact, in order to reveal the fault (i.e., kill the mutant), we need the binary search to fail because the second set is unordered in memory. Since characterizing this condition would take too much time and space in this thesis, we do not further discuss the condition under which the mutant would be killed. It is sufficient to say that, at this stage of the discussion, it is difficult to imagine a CP specification that would necessarily lead to exercising this condition. This therefore appears to be simply an input selection issue.

All the test suites achieve 100% coverage of the `difference` method for the Statement, Branch and Term criteria. 25 mutants were generated for the difference method. 96% of the mutants are killed by all test suites while 4% (one) remain alive. The mutant that remains alive is equivalent. The CP specification being specifically focusing on the difference function, we find no difference between the selection criteria in terms of effectiveness.

```
AOIS_41:38:int_make_a_free_slot(int):i => ++i
AOIS_42:38:int_make_a_free_slot(int):i => --i
AOIS_43:38:int_make_a_free_slot(int):i => i++
AOIS_44:38:int_make_a_free_slot(int):i => i--
AOIS_45:38:int_make_a_free_slot(int):_last => ++_last
AOIS_46:38:int_make_a_free_slot(int):_last => --_last
AOIS_47:38:int_make_a_free_slot(int):_last => _last++
AOIS_48:38:int_make_a_free_slot(int):_last => _last--
AOIS_49:38:int_make_a_free_slot(int):_last => ++_last
AOIS_50:38:int_make_a_free_slot(int):_last => --_last
AOIS_51:38:int_make_a_free_slot(int):_last => _last++
AOIS_52:38:int_make_a_free_slot(int):_last => _last-
```

AOIS_41 to AOIS_45, AOIS_47, AOIS_49 and AOIS_51 are killed by all test suites.

Changing the value of i will cause the if statement to return true when it should return false and vice versa. This is why all mutants that change the value of i are killed by all test suites.

AOIS_46, AOIS_48, AOIS_50 and AOIS_52 are killed by any of the test suites. This is because changing the value of last will cause no items to be added to the sets. When compared to the actual difference set, the program returns false.

```
AOIS_53:39:int_make_a_free_slot(int):_last => ++_last
AOIS_54:39:int_make_a_free_slot(int):_last => --_last
```

```
AOIS_55:39:int_make_a_free_slot(int):_last => _last++
AOIS_56:39:int_make_a_free_slot(int):_last => _last--
AOIS_57:39:int_make_a_free_slot(int):j => ++j
AOIS_58:39:int_make_a_free_slot(int):j => --j
AOIS_59:39:int_make_a_free_slot(int):j => j++
AOIS_60:39:int_make_a_free_slot(int):j => j--
AOIS_61:39:int_make_a_free_slot(int):i => ++i
AOIS_62:39:int_make_a_free_slot(int):i => --i
AOIS_63:39:int_make_a_free_slot(int):i => i++
AOIS_64:39:int_make_a_free_slot(int):i => i-
```

If the program enters the if statement, the value of last is increased thereby increasing the position of where values are added to the set. This causes the program to try and add more items to the set than the capacity. Therefore AOIS_53 and AOIS_55 are killed by all test suites.

AOIS_54 and AOIS_56 are killed by any test suite. Decreasing the value of last will cause no items to be added to the sets. When comparing to the actual difference set that is non-empty, these mutants will be killed. Since all test suites contain a choice with non-empty difference set, these mutants are killed.

AOIS_57 and AOIS_61 to AOIS_64 are killed by all test suites.

Increasing the value of j will cause the program to go into an infinite loop, and therefore

Changing i will cause the program to go into an infinite loop. Therefore AOIS_61 to AOIS_64 are killed by all test suites.

AOIS_58 to AOIS_60 are not killed by the Base Choice and Pair Wise test suites but are killed by the Each Choice test suite. Since the Each Choice test suite is the only test suite that reaches this part of the code these mutants are killed by this test suite. The Each Choice test suite contains a test case where the values of the set are unordered.

```
AOIS_65:40:int_make_a_free_slot(int):j => ++j
AOIS_66:40:int_make_a_free_slot(int):j => --j
AOIS_67:40:int_make_a_free_slot(int):j => j++
AOIS_68:40:int_make_a_free_slot(int):j => j-
```

AOIS_66 and AOIS_68 are not killed by the Base Choice and Pair Wise test suites but are killed by the Each Choice test suite. This is because of the same reason as AOIS_58 to AOIS_60.

```
AOIS_69:42:int_make_a_free_slot(int):i => ++i
AOIS_70:42:int_make_a_free_slot(int):i => --i
AOIS_71:42:int_make_a_free_slot(int):i => i++
AOIS_72:42:int_make_a_free_slot(int):i => i-
```

AOIS_69 increases the value of pos. Since the Pair Wise and Base Choice test suites contain a test case where there is only one value in the difference set, the program tries to access position one in the set which does not exist. Therefore this mutant is killed by these two test suites. The Each Choice test suite is also able to kill this mutant because it contains an unordered set. This change causes the set to remain unordered and as a result the comparison to the correct ordered difference set fails resulting in killing of mutant.

The `pos` value is decreased by one in this mutant. Since initial value of `_last` is -1, the program decreases the value of `pos` by one causing the first element of the set to not be added to the set. This causes the difference set to different since the first item is not added. Therefore AOIS_70 is killed by all test suites.

AOIS_71 and AOIS_72 change i after assignment and the value of position is still equal to i. Since i is never used after this assignment, these mutants are considered to be equivalent and not killed by any of the test suites.

```
AOIS_73:44:int_make_a_free_slot(int):_last => ++_last
AOIS_74:44:int_make_a_free_slot(int):_last => --_last
AOIS_75:44:int_make_a_free_slot(int):_last => _last++
AOIS_76:44:int_make_a_free_slot(int):_last => _last-
```

AOIS_73 and AOIS_75 increase the value of last by one. This change causes the last element to not be added to the set because the set becomes full. These two mutants are killed by all test suites.

AOIS_74 is killed by all test suites. This mutant causes the items to be added to the first position in an ordered set leaving only the last item in the set. Since only the last item is added to a set, if the first set contains values that are not in the second set, this mutant is caught

AOIS_76 is killed by all test suites. This mutant does the opposite of AOIS_74. Only the first item is added. If a test case contains sets that have the first element of the first set in the second set this mutant is caught. Since all test suites contain such test case, this mutant is killed by all.

```
AOIS_77:47:int_make_a_free_slot(int):pos => pos++
AOIS_78:47:int_make_a_free_slot(int):pos => pos-
```

These mutants are not killed by any of the test suites. Since the value of pos is not used after the return statement, these mutants are equivalent.

```
AOIS_79:53:void_addElement(int):n => ++n
AOIS_80:53:void_addElement(int):n => --n
AOIS_81:53:void_addElement(int):n => n++
AOIS_82:53:void_addElement(int):n => n-
```

These mutants are killed by all test suites because changing the value of n will cause the wrong value to be added to the set.

```
AOIS_83:57:void_addElement(int):_last => ++_last
AOIS_84:57:void_addElement(int):_last => --_last
AOIS_85:57:void_addElement(int):_last => _last++
AOIS_86:57:void_addElement(int):_last => _last--
AOIS_87:57:void_addElement(int):_set_size => _set_size++
AOIS_88:57:void_addElement(int):_set_size => _set_size-
```

AOIS_83, AOIS_84, AOIS_85, AOIS_86 and AOIS_88 are killed by all test suites. Increasing the value of last will cause the program not to add elements because the set is assumed to be full since last is increased by one. Decreasing the size of last after the comparison will cause the program to not add the last element in the set because the size is decreased.

AOIS_87 is not killed by any of the test suites. Since we are not adding more elements than the set's size, this mutant is not killed.

```
AOIS_89:58:void_addElement(int):n => ++n
AOIS_90:58:void_addElement(int):n => --n
AOIS_91:58:void_addElement(int):n => n++
AOIS_92:58:void_addElement(int):n => n-
```

These mutants are not killed by any of the test suites. These changes are in a print statement which has no effect on the functionality.

```
AOIS_93:62:void_addElement(int):n => ++n
AOIS_94:62:void_addElement(int):n => --n
AOIS_95:62:void_addElement(int):n => n++
AOIS_96:62:void_addElement(int):n => n-
```

Changing the value of n will cause improper elements to be added to the set. therefore these mutants are killed by all test suites.

```
AOIS_97:63:void_addElement(int):pos => ++pos
AOIS_98:63:void_addElement(int):pos => --pos
AOIS_99:63:void_addElement(int):pos => pos++
AOIS_100:63:void_addElement(int):pos => pos-
```

Changing the value of position will cause the program to try and access a value that does not exist in the set. This is why these mutants are killed by all test suites.

```
AOIS_101:64:void_addElement(int):n => ++n
AOIS_102:64:void_addElement(int):n => --n
AOIS_103:64:void_addElement(int):n => n++
AOIS_104:64:void_addElement(int):n => n-
```

AOIS_101 and AOIS_102 are killed by all test suites. This causes invalid values to be added to the set therefore they are killed by all test suites.

AOIS_103 and AOIS_104 are not killed by any test suite. since n is never used after this statement, changing n after comparison will not have an effect on the functionality. Therefore these mutants are equivalent and not killed by any of the test suites.

```
AOIS_105:72:int_getSize():_last => ++_last
AOIS_106:72:int_getSize():_last => --_last
AOIS_107:72:int_getSize():_last => _last++
AOIS_108:72:int_getSize():_last => _last-
```

These mutants are killed by all test suites. Changing the values of last will cause the wrong value to be returned and this causes array out of bound exceptions when adding elements to the set.

```
AOIS_109:77:int_getElementAt(int):i => ++i
AOIS_110:77:int_getElementAt(int):i => --i
AOIS_111:77:int_getElementAt(int):i => i++
AOIS_112:77:int_getElementAt(int):i => i--
AOIS_113:77:int_getElementAt(int):i => ++i
AOIS_114:77:int_getElementAt(int):i => --i
AOIS_115:77:int_getElementAt(int):i => i++
AOIS_116:77:int_getElementAt(int):i => i--
AOIS_117:77:int_getElementAt(int):_last => ++_last
AOIS_118:77:int_getElementAt(int):_last => --_last
AOIS_119:77:int_getElementAt(int):_last => _last++
AOIS_120:77:int_getElementAt(int):_last => _last-
```

AOIS_109 to AOIS_116, AOIS_118 and AOIS_120 are killed by all test suites. Changing i will have an effect on the functionality of the program and therefore these mutants are killed.

AOIS_117 and AOIS_119 are not killed by any of the sets. Since we are not adding more elements than the capacity of the set, these mutants are not killed.

```
AOIS_121:80:int_getElementAt(int):i => i++
AOIS_122:80:int_getElementAt(int):i => i-
```

These mutants are not killed by any of the test suites because i is never used after this line.

```
AOIS_123:87:int_binSearch(int):_last => _last++
AOIS_124:87:int_binSearch(int):_last => _last-
```

These mutants are killed by all test suites. Increasing the value of _last will cause the program to not find items in the list and it also increases the loop by one iteration. AOIS_124 is killed by all test suites because this change causes no items to be added to any of the sets. The difference set is empty as a result and when compared to the actual difference set, the program returns false.

```
AOIS_125:89:int_binSearch(int):_last => ++_last
```

```
AOIS_126:89:int_binSearch(int):_last => --_last
AOIS_127:89:int_binSearch(int):_last => _last++
AOIS_128:89:int_binSearch(int):_last => _last--
```

AOIS_125 and AOIS_127 are killed by all test suites. This is because of the same reason as AOIS_123.

AOIS_126 and AOIS_128 are killed by all test suites because of the same reason as AOIS_124.

```
AOIS_129:92:int_binSearch(int):i => ++i
AOIS_130:92:int_binSearch(int):i => --i
AOIS_131:92:int_binSearch(int):i => i++
AOIS_132:92:int_binSearch(int):i => i--
AOIS_133:92:int_binSearch(int):j => ++j
AOIS_134:92:int_binSearch(int):j => --j
AOIS_135:92:int_binSearch(int):j => j++
AOIS_136:92:int_binSearch(int):j => j-
```

These mutants are killed by all test suites. These changes cause the loop to not complete the number of iterations or have more iterations than expected. They also cause items to not be found and therefore they are killed by all test suites.

```
AOIS_137:93:int_binSearch(int):i => ++i
AOIS_138:93:int_binSearch(int):i => --i
AOIS_139:93:int_binSearch(int):i => i++
AOIS_140:93:int_binSearch(int):i => i--
AOIS_141:93:int_binSearch(int):j => ++j
AOIS_142:93:int_binSearch(int):j => --j
AOIS_143:93:int_binSearch(int):j => j++
AOIS_144:93:int_binSearch(int):j => j-
AOIS_137 to AOIS_143 are killed by all test suites.
```

AOIS_144 is not killed by the Each Choice test suite but killed by the Base Choice and Pair Wise test suites. Since the boundary's value is decreased, if the item in the position of _last/2 is present in both sets, this item is missed and therefore the mutant is caught. The Base Choice and Pair Wise test suite contain such item and are therefore able to kill this mutant.

```
AOIS_145:94:int_binSearch(int):x => ++x
AOIS_146:94:int_binSearch(int):x => --x
AOIS_147:94:int_binSearch(int):x => x++
AOIS_148:94:int_binSearch(int):x => x--
AOIS_149:94:int_binSearch(int):m => ++m
AOIS_150:94:int_binSearch(int):m => --m
AOIS_151:94:int_binSearch(int):m => m++
AOIS_152:94:int_binSearch(int):m => m-
```

Changing these values causes some items in the set to be missed by the search. Therefore they are killed by all test suites.

```
AOIS_153:95:int_binSearch(int):m => ++m
AOIS_154:95:int_binSearch(int):m => --m
```

```
AOIS_155:95:int_binSearch(int):m => m++
AOIS_156:95:int_binSearch(int):m => m--
AOIS_157:97:int_binSearch(int):m => ++m
AOIS_158:97:int_binSearch(int):m => --m
AOIS_159:97:int_binSearch(int):m => m++
AOIS_160:97:int_binSearch(int):m => m-
```

AOIS_153, AOIS_154, AOIS_157 and AOIS_158 are killed by all test suites. Changing the value of m will cause some values to be missed.

AOIS_155, AOIS_156, AOIS_159 and AOIS_160 are not killed by any test suite. The value of m is not used again after this line. Increasing or decreasing the value of m will therefore not change the functionality. These four mutants are equivalent and not killed by any of the test suites.

```
AOIS_161:100:int_binSearch(int):x => ++x
AOIS_162:100:int_binSearch(int):x => --x
AOIS_163:100:int_binSearch(int):x => x++
AOIS_164:100:int_binSearch(int):x => x--
AOIS_165:100:int_binSearch(int):i => ++i
AOIS_166:100:int_binSearch(int):i => --i
AOIS_167:100:int_binSearch(int):i => i++
AOIS_168:100:int_binSearch(int):i => i-
```

AOIS_161, AOIS_162, AOIS_165, AOIS_166 and AOIS_168 are killed by all test suites because changing these values has an impact on the functionality and all test suites contain a choice that is non-empty.

AOIS_163, AOIS_164 and AOIS_167 are not killed by any of the test suites. Changing the value of x after the evaluation will not have an impact on the functionality because x is never used after the statement. Therefore AOIS_163 and AOIS_164 are equivalent.

If the program reaches the return statement, the value of i returned will be increased by 1. Since there are no duplicate values and this method is only used for the difference function, the actual value of i will not be used and it is only important if it is greater than zero. Therefore this is not killed by any of the test suites.

```
AOIS_169:101:int_binSearch(int):i => i++
AOIS_170:101:int_binSearch(int):i => i--
```

These mutants are not killed by any of the test suites. This change will not change the functionality because it is in the return statement and the value of i returned will be the same. These mutants are equivalent.

```
AOIS_171:112:OrdSetSimple_difference(OrdSetSimple):size1 => size1++
```

```
AOIS_172:112:OrdSetSimple_difference(OrdSetSimple):size1 => size1-
```

Increasing the size will cause the loop to be executed one extra time which will result in item not found and not added to the difference set. AOIS_171 is not killed by any test suite. Decreasing the size will cause the last item not to be added to the difference set. Since all test suites contain a test case where the last value is being added to the difference set, AOIS_172 is killed by all test suites.

```
AOIS_173:114:OrdSetSimple_difference(OrdSetSimple):k => ++k
AOIS_174:114:OrdSetSimple_difference(OrdSetSimple):k => --k
AOIS_175:114:OrdSetSimple_difference(OrdSetSimple):k => k++
AOIS_176:114:OrdSetSimple_difference(OrdSetSimple):k => k--
AOIS_177:114:OrdSetSimple_difference(OrdSetSimple):size1 => ++size1
AOIS_178:114:OrdSetSimple_difference(OrdSetSimple):size1 => --size1
AOIS_179:114:OrdSetSimple_difference(OrdSetSimple):size1 => size1++
AOIS_180:114:OrdSetSimple_difference(OrdSetSimple):size1 => size1--
```

AOIS_173 to AOIS_176 are killed by all test suites since changing the value of k will cause some elements to not be compared and missed. This is the same case for AOIS_177 to AOIS_180 with the exception of size being changed instead of k.

```
AOIS_181:115:OrdSetSimple_difference(OrdSetSimple):k => ++k
AOIS_182:115:OrdSetSimple_difference(OrdSetSimple):k => --k
AOIS_183:115:OrdSetSimple_difference(OrdSetSimple):k => k++
AOIS_184:115:OrdSetSimple_difference(OrdSetSimple):k => k--
```

These mutants are killed by all test suites because of the same reason as AOIS_173 to AOIS_176.

```
AOIS_185:116:OrdSetSimple_difference(OrdSetSimple):k => k++
AOIS_186:116:OrdSetSimple_difference(OrdSetSimple):k => k--
```

These mutants are killed by all test suites because of the same reason as AOIS_173 to AOIS_176.

```
AOIS_187:126:java.lang.String_toString():k => ++k
AOIS_188:126:java.lang.String_toString():k => --k
AOIS_189:126:java.lang.String_toString():k => k++
AOIS_190:126:java.lang.String_toString():k => k--
AOIS_191:126:java.lang.String_toString():_last => ++_last
AOIS_192:126:java.lang.String_toString():_last => --_last
AOIS_193:126:java.lang.String_toString():_last => _last++
AOIS_194:126:java.lang.String_toString():_last => _last--
```

AOIS_187, AOIS_192 and AOIS_194 are not killed by any test suite.

Increasing the value of k before comparison will cause the program not to enter the loop and all the toString methods will return an empty string. When comparing these values the comparison returns true since two strings being compared are equal. Therefore AOIS_187 is not killed by

any of the test suites. AOIS_188 to AOIS_191 are killed. Decreasing the value of k causes the program to try to access the set at -1 (k starts at 0 and is decreased to -1). This causes an array out of bound exception. Therefore AOIS_188 and AOIS_190 are killed by all test suites. Increasing the value of k after the first comparison will cause the program to try and access the set at the size of set plus one. This causes array out of bound exception as well. Therefore AOIS_189 is killed by all test suites. The same rational can be applied for AOIS_191 to AOIS_194. AOIS_191 and AOIS_193 are killed by all test suites. AOIS_192 and AOIS_194 are not killed by any of the test suites.

```
AOIS_195:127:java.lang.String_toString():k => ++k
AOIS_196:127:java.lang.String_toString():k => --k
AOIS_197:127:java.lang.String_toString():k => k++
AOIS_198:127:java.lang.String_toString():k => k--
```

AOIS_195, AOIS_196 and AOIS_198 are killed by all test suites.

AOIS_197 is not killed by any of the test suites because it eliminates one of the items from the end of the set.

The other mutants in this section are killed by all test suites. These mutants cause the program to try and access a value of an array that does not exist. This causes an array out of bound exception and therefore killed by all test suites.

```
AOIS_199:148:boolean_equals(java.lang.Object):i => ++i
AOIS_200:148:boolean_equals(java.lang.Object):i => --i
AOIS_201:148:boolean_equals(java.lang.Object):i => i++
AOIS_202:148:boolean_equals(java.lang.Object):i => i--
AOIS_203:149:boolean_equals(java.lang.Object):i => ++i
AOIS_204:149:boolean_equals(java.lang.Object):i => --i
AOIS_205:149:boolean_equals(java.lang.Object):i => i++
AOIS_206:149:boolean_equals(java.lang.Object):i => i--
AOIS_207:149:boolean_equals(java.lang.Object):i => ++i
AOIS_208:149:boolean_equals(java.lang.Object):i => --i
AOIS_209:149:boolean_equals(java.lang.Object):i => i++
AOIS_210:149:boolean_equals(java.lang.Object):i => i--
```

These mutants are not killed by any of the test suites because the equals method is never used by any test case.

```
ROR_1:30:int_make_a_free_slot(int): pos >= 0  =>   pos > 0
ROR_2:30:int_make_a_free_slot(int): pos >= 0  =>   pos < 0
ROR_3:30:int_make_a_free_slot(int): pos >= 0  =>   pos <= 0
ROR_4:30:int_make_a_free_slot(int): pos >= 0  =>   pos == 0
ROR_5:30:int_make_a_free_slot(int): pos >= 0  =>   pos != 0
```

ROR_1 and ROR_4 are not killed by any of the test suites. This is because there are no test cases with redundant values therefore the value of position will always be less than 0.

ROR_2, ROR_3 and ROR_5 are killed by all test suites. By entering the loop the program will always return 1, therefore it replaces the set at position 1 with the given value. Comparing the returned value to the actual value, the program returns false and these mutants are therefore caught.

```
ROR_6:33:int_make_a_free_slot(int): i <= _last  =>   i > _last
ROR_7:33:int_make_a_free_slot(int): i <= _last  =>   i >= _last
ROR_8:33:int_make_a_free_slot(int): i <= _last  =>   i < _last
ROR_9:33:int_make_a_free_slot(int): i <= _last  =>   i == _last
ROR_10:33:int_make_a_free_slot(int): i <= _last  =>   i != _last
```

These mutants are killed by all test suites. Since all test suites contain a test case with non-empty values, and reach this part of the code. These mutants are killed. Changing the <= to any other value will cause the program to enter the loop when it should not and not enter the loop when it should.

```
ROR_11:34:int_make_a_free_slot(int): _set[i] > val  =>   _set[i] >= val
ROR_12:34:int_make_a_free_slot(int): _set[i] > val  =>   _set[i] < val
ROR_13:34:int_make_a_free_slot(int): _set[i] > val  =>   _set[i] <= val
ROR_14:34:int_make_a_free_slot(int): _set[i] > val  =>   _set[i] == val
ROR_15:34:int_make_a_free_slot(int): _set[i] > val  =>   _set[i] != val
```

ROR_11 is not killed by any of the test suites. The set does not contain duplicate values and therefore will never have a value that is equal to val. This is why ROR_11 is not killed by any of the test suites.

ROR_12, ROR_13 and ROR_15 are killed by all test suites. This is because changing this value will cause the program to go to the `break` and the set to be in reverse order, which is not an ordered set.

ROR_14 is not killed by the Base Choice and Pair Wise test suites but is killed by the Each Choice test suite. Since only the Each Choice test suite has an unordered set, this change will cause the program to not go into this line, which means the set will not be ordered. Therefore when sets s1 and s2 are compared they result will not be correct.

```
ROR_16:38:int_make_a_free_slot(int): i <= _last  =>   i > _last
ROR_17:38:int_make_a_free_slot(int): i <= _last  =>   i >= _last
ROR_18:38:int_make_a_free_slot(int): i <= _last  =>   i < _last
ROR_19:38:int_make_a_free_slot(int): i <= _last  =>   i == _last
ROR_20:38:int_make_a_free_slot(int): i <= _last  =>   i != _last
```

```
ROR_21:38:int_make_a_free_slot(int): _last == -1  =>   _last > -1
ROR_22:38:int_make_a_free_slot(int): _last == -1  =>   _last >= -1
ROR_23:38:int_make_a_free_slot(int): _last == -1  =>   _last < -1
ROR_24:38:int_make_a_free_slot(int): _last == -1  =>   _last <= -1
ROR_25:38:int_make_a_free_slot(int): _last == -1  =>   _last != -1
```

ROR_16 to ROR_20 are not killed by the Base Choice and Pair Wise test suites, but are killed by the Each Choice test suite. This change causes the unordered set to remain unordered, and since the only test suite that contains an unordered set is the Each Choice test suite, it is able to kill these mutants.

ROR_21 to ROR_25 are not killed by any of the test suites. If the value of the operator is changed and the program does not reach this statement, the position will be set to _last+1, which is equivalent. Therefore this mutant is not killed by any of the test suites.

```
ROR_26:39:int_make_a_free_slot(int): j >= i  =>   j > i
ROR_27:39:int_make_a_free_slot(int): j >= i  =>   j < i
ROR_28:39:int_make_a_free_slot(int): j >= i  =>   j <= i
ROR_29:39:int_make_a_free_slot(int): j >= i  =>   j == i
ROR_30:39:int_make_a_free_slot(int): j >= i  =>   j != i
```

ROR_26 and ROR_29 are killed by the Each Choice test suite but not by the Pair Wise and Base Choice test suites. Since only the Each Choice has an unordered set, changing the value from > to >= and == will only change the functionality for this test suite. The other test suites are already ordered and will never have such case.

ROR_27, ROR_28 and ROR_30 are killed by all test suites. Changing this value will cause the program to put the values in reverse order. This causes the final set difference value to not be equal and therefore is killed by all test suites.

```
ROR_31:57:void_addElement(int): _last + 1 >= _set_size  =>   _last + 1 >
_set_size
ROR_32:57:void_addElement(int): _last + 1 >= _set_size  =>   _last + 1 <
_set_size
ROR_33:57:void_addElement(int): _last + 1 >= _set_size  =>   _last + 1 <=
_set_size
ROR_34:57:void_addElement(int): _last + 1 >= _set_size  =>   _last + 1 ==
_set_size
ROR_35:57:void_addElement(int): _last + 1 >= _set_size  =>   _last + 1 !=
_set_size
```

ROR_31 and ROR_34 are not killed by any of the test suites because we never add values to a set that is full.

ROR_32, ROR_33 and ROR_35 are killed by all test suites because this change causes no values to be added to sets resulting in an empty difference set. Since all test suites contain test cases

where the difference set is not empty, when compared to the actual difference set the program returns false and results in mutant being caught.

```
ROR_36:63:void_addElement(int): pos >= 0  =>   pos > 0
ROR_37:63:void_addElement(int): pos >= 0  =>   pos < 0
ROR_38:63:void_addElement(int): pos >= 0  =>   pos <= 0
ROR_39:63:void_addElement(int): pos >= 0  =>   pos == 0
ROR_40:63:void_addElement(int): pos >= 0  =>   pos != 0
```

ROR_36 to ROR_40 are killed by all test suites because changing these values will cause no values to be added to the sets and the difference set to be empty.

```
ROR_41:77:int_getElementAt(int): i > _last  =>   i >= _last
ROR_42:77:int_getElementAt(int): i > _last  =>   i < _last
ROR_43:77:int_getElementAt(int): i > _last  =>   i <= _last
ROR_44:77:int_getElementAt(int): i > _last  =>   i == _last
ROR_45:77:int_getElementAt(int): i > _last  =>   i != _last
```

ROR_41 will cause the program to return -1 for get element at for the last item on the list. This will cause the program to try and add -1 which is impossible. Therefore the last item will always be missed. If the last item in the first list is contained in the difference set (which it is in at least one test case for every test suite) this mutant will be killed. Therefore this mutant is killed by all test suites.

ROR_44 will cause the program to return -1 for the last item as well. Therefore this mutant is killed by all test suites.

ROR_42, ROR_44 and ROR_45 are killed by all test suites. ROR_42 causes the program to return -1 when the item being retrieved is less than the size of the set. This is true all the time when the sets are not empty. Therefore this mutant is killed by all test suites. The same logic applies for ROR_44 and ROR_45.

```
ROR_46:89:int_binSearch(int): _last == -1  =>   _last > -1
ROR_47:89:int_binSearch(int): _last == -1  =>   _last >= -1
ROR_48:89:int_binSearch(int): _last == -1  =>   _last < -1
ROR_49:89:int_binSearch(int): _last == -1  =>   _last <= -1
ROR_50:89:int_binSearch(int): _last == -1  =>   _last != -1
```

ROR_49 is not killed by any of the test suites. The value of the integer looked for in binary search is never less than -1, this is why ROR_49 is not killed by any of the test suites. This mutant is equivalent

Changing the operators will cause the program to return -1 when last is not equal to -1, this means that items are not found when they do infact exist. This is why ROR_46 to ROR_48 and ROR_50 are killed by all test suites.

```
ROR_51:94:int_binSearch(int): x > _set[m]  =>   x >= _set[m]
ROR_52:94:int_binSearch(int): x > _set[m]  =>   x < _set[m]
ROR_53:94:int_binSearch(int): x > _set[m]  =>   x <= _set[m]
ROR_54:94:int_binSearch(int): x > _set[m]  =>   x == _set[m]
ROR_55:94:int_binSearch(int): x > _set[m]  =>   x != _set[m]
```

Changing the operator will cause the program to not find the values in the set. Since all test suites contain a test case where the first set contains items that are not in the second set, these mutants are killed by all test suites.

```
ROR_56:100:int_binSearch(int): x == _set[i]  =>   x > _set[i]
ROR_57:100:int_binSearch(int): x == _set[i]  =>   x >= _set[i]
ROR_58:100:int_binSearch(int): x == _set[i]  =>   x < _set[i]
ROR_59:100:int_binSearch(int): x == _set[i]  =>   x <= _set[i]
ROR_60:100:int_binSearch(int): x == _set[i]  =>   x != _set[i]
```

ROR_56, ROR_58 and ROR_60 are killed by all test suites. When the operator is changed it will cause the item to be marked as not found when it is found and vice versa. Since all test suites contain a choice that is non-empty these mutants are killed.

ROR_59 is not killed by Pair Wise test suite but is killed by all other test suites. Pair Wise test suite only contains a test case where only the last item in the first set is not contained in the second set.

```
ROR_61:126:java.lang.String_toString(): k <= _last  =>   k > _last
ROR_62:126:java.lang.String_toString(): k <= _last  =>   k >= _last
ROR_63:126:java.lang.String_toString(): k <= _last  =>   k < _last
ROR_64:126:java.lang.String_toString(): k <= _last  =>   k == _last
ROR_65:126:java.lang.String_toString(): k <= _last  =>   k != _last
```

ROR_63 and ROR_64 are not killed by any of the test suites. If the operator is changed to ==, the program will return a null string since the first value of k is 0 and the value of _last does not equal to k, this statement will always return false. Since this is being compared to a null string, the value returned is true. ROR_42 will cause the last item of the set to not be added to the string. This is the same case for the string that it is being compared to. Therefore this mutant is not killed by any of the test suites.

ROR_61, ROR_62 and ROR_65 are killed by all test suites. An empty set will never have a k value that is equal to _last or less than last. This causes the program to keep increasing k and

adding zero until it cannot add any more values due to array out of bounds exception. Since all test suites have a test case with an empty difference set, these mutants are killed by all.

```
ROR_66:134:boolean_equals(java.lang.Object): this == obj  =>   this != obj
This mutant is not killed by any of the test suites.

ROR_67:140:boolean_equals(java.lang.Object): this == null  =>   this !=
null
ROR_68:140:boolean_equals(java.lang.Object): null == obj  =>   null != obj
ROR_69:145:boolean_equals(java.lang.Object): this.getSize() == 0  =>
this.getSize() > 0
ROR_70:145:boolean_equals(java.lang.Object): this.getSize() == 0  =>
this.getSize() >= 0
ROR_71:145:boolean_equals(java.lang.Object): this.getSize() == 0  =>
this.getSize() < 0
ROR_72:145:boolean_equals(java.lang.Object): this.getSize() == 0  =>
this.getSize() <= 0
ROR_73:145:boolean_equals(java.lang.Object): this.getSize() == 0  =>
this.getSize() != 0
ROR_74:145:boolean_equals(java.lang.Object): ph.getSize() == 0  =>
ph.getSize() > 0
ROR_75:145:boolean_equals(java.lang.Object): ph.getSize() == 0  =>
ph.getSize() >= 0
ROR_76:145:boolean_equals(java.lang.Object): ph.getSize() == 0  =>
ph.getSize() < 0
ROR_77:145:boolean_equals(java.lang.Object): ph.getSize() == 0  =>
ph.getSize() <= 0
ROR_78:145:boolean_equals(java.lang.Object): ph.getSize() == 0  =>
ph.getSize() != 0
ROR_79:149:boolean_equals(java.lang.Object): this.getElementAt( i ) !=
ph.getElementAt( i )  =>   this.getElementAt( i ) > ph.getElementAt( i )
ROR_80:149:boolean_equals(java.lang.Object): this.getElementAt( i ) !=
ph.getElementAt( i )  =>   this.getElementAt( i ) >= ph.getElementAt( i )
ROR_81:149:boolean_equals(java.lang.Object): this.getElementAt( i ) !=
ph.getElementAt( i )  =>   this.getElementAt( i ) < ph.getElementAt( i )
ROR_82:149:boolean_equals(java.lang.Object): this.getElementAt( i ) !=
ph.getElementAt( i )  =>   this.getElementAt( i ) <= ph.getElementAt( i )
ROR_83:149:boolean_equals(java.lang.Object): this.getElementAt( i ) !=
ph.getElementAt( i )  =>   this.getElementAt( i ) == ph.getElementAt( i )
ROR_84:154:boolean_equals(java.lang.Object): this.getSize() ==
ph.getSize()  =>   this.getSize() > ph.getSize()
ROR_85:154:boolean_equals(java.lang.Object): this.getSize() ==
ph.getSize()  =>   this.getSize() >= ph.getSize()
ROR_86:154:boolean_equals(java.lang.Object): this.getSize() ==
ph.getSize()  =>   this.getSize() < ph.getSize()
ROR_87:154:boolean_equals(java.lang.Object): this.getSize() ==
ph.getSize()  =>   this.getSize() <= ph.getSize()
ROR_88:154:boolean_equals(java.lang.Object): this.getSize() ==
ph.getSize()  =>   this.getSize() != ph.getSize()
```

These mutants are not killed by any of the test suites because the equals method is never used.

```
COR_1:38:int_make_a_free_slot(int): i <= _last || _last == -1  =>   i <=
_last && _last == -1
```

```
COR_2:38:int_make_a_free_slot(int): i <= _last || _last == -1  =>   i <=
_last ^ _last == -1
```

COR_1 is killed by the Each Choice test suite because this test suite contains a test case where the set is unordered and therefore enters this statement. Since both conditions cannot be true, this statement returns false instead of true and vice versa.

COR_2 is not killed by any of the test suites because in this case having OR is the same as an exclusive or. This mutant is equivalent.

```
COR_3:77:int_getElementAt(int): i < 0 || i > _last  =>   i < 0 && i >
_last
COR_4:77:int_getElementAt(int): i < 0 || i > _last  =>   i < 0 ^ i > _last
```

These mutants are not killed by any of the test suites. There are no test cases where either of the conditions are true, therefore these mutants are not killed.

```
COR_5:140:boolean_equals(java.lang.Object): this == null || null == obj
=>   this == null && null == obj
COR_6:140:boolean_equals(java.lang.Object): this == null || null == obj
=>   this == null ^ null == obj
COR_7:145:boolean_equals(java.lang.Object): this.getSize() == 0 &&
ph.getSize() == 0  =>   this.getSize() == 0 || ph.getSize() == 0
COR_8:145:boolean_equals(java.lang.Object): this.getSize() == 0 &&
ph.getSize() == 0  =>   this.getSize() == 0 ^ ph.getSize() == 0
COD_1:137:boolean_equals(java.lang.Object):!(obj instanceof OrdSetSimple)
=> obj instanceof OrdSetSimple
COI_1:18:OrdSetSimple(int): k < _set_size  =>  !(k < _set_size)
for (k = 0; k < _set_size; k++) {
```

These mutants are not killed by any of the test suites because the `equals` method is never used.

```
COI_2:30:int_make_a_free_slot(int): pos >= 0  =>  !(pos >= 0)
```

This mutant is killed by all test suites. Since the value of position will cause the values to be added to position 1 in the set, this will throw an array out of bound exception.

```
COI_3:33:int_make_a_free_slot(int): i <= _last  =>  !(i <= _last)
```

This change will cause the program to enter this loop when there are no items in the set. This causes an array out of bound exception and is therefore killed by all test suites.

COI_4:34:int_make_a_free_slot(int): _set[i] > val => !(_set[i] > val)

This change will cause the set to be ordered in reverse. Therefore when compared to an ordered set the result is false. Therefore this mutant is killed by all test suites.

```
COI_5:38:int_make_a_free_slot(int): i <= _last  =>  !(i <= _last)
COI_6:38:int_make_a_free_slot(int): _last == -1  =>  !(_last == -1)
```

```
COI_7:38:int_make_a_free_slot(int): i <= _last || _last == -1  =>  !(i <=
_last || _last == -1)
```

COI_5 and COI_7 are killed by the Each Choice but not by the Base Choice test suite. The changes made by these two mutants cause the program to go into the loop but for an ordered set this will not make a difference. However if the set is not ordered this will cause the set to remain unordered and therefore when being compared to result it will not be equal. Therefore this mutant is killed by the Each Choice test suite since it is the only test suite that contains an unordered set.

COI_6 is not killed by any of the test suites. If the value of last is -1, pos will be 0 no matter if the program enters this part of the program or the else statement. Therefore this mutant is equivalent and not killed by any of the test suites.

```
COI_8:39:int_make_a_free_slot(int): j >= i  =>  !(j >= i)
```

This change will cause the program to enter the loop when the value of _last is -1. This causes an array out of bound exception and is killed by all test suites.

```
COI_9:53:void_addElement(int): n < 0  =>  !(n < 0)
```

This change will cause the program not to add any value to the sets. Since all test suites contain a test case where the difference set is not empty, when compared to the actual difference set the program returns false. Therefore this mutant is killed by all test suites.

```
COI_10:57:void_addElement(int): _last + 1 >= _set_size  =>  !(_last + 1 >=
_set_size)
```

This mutant is killed by all test suites because of the same reason as COI_9.

```
COI_11:63:void_addElement(int): pos >= 0  =>  !(pos >= 0)
```

This change causes the program to access a value of -1 which causes an array out of bounds exception. Therefore this mutant is killed by all test suites.

```
COI_12:77:int_getElementAt(int): i < 0  =>  !(i < 0)
COI_13:77:int_getElementAt(int): i > _last  =>  !(i > _last)
COI_14:77:int_getElementAt(int): i < 0 || i > _last  =>  !(i < 0 || i >
_last)
```

These mutants cause the program to enter this part of the code even if the item exists in the set. this causes the program to try and add -1 to the set instead of correct values. This means that the resulting difference set is all zeros. Therefore this mutant is killed because all test suites contain a test case where the difference set is not empty and the explanation provided above.

```
COI_15:89:int_binSearch(int): _last == -1  =>  !(_last == -1)
```

This mutant causes the program to return -1 even if the set is not empty. Therefore this is killed by all test suites.

```
COI_16:92:int_binSearch(int): i < j  =>  !(i < j)
```

This mutant is killed by all test suites because this change will also cause the program to return -1 even if the set is not empty.

```
COI_17:94:int_binSearch(int): x > _set[m]  =>  !(x > _set[m])
COI_18:100:int_binSearch(int): x == _set[i]  =>  !(x == _set[i])
```

These mutants are killed by all test suites because they change the functionality.

```
COI_19:114:OrdSetSimple_difference(OrdSetSimple): k < size1  =>  !(k <
size1)
```

This change will cause the program to go into an infinite loop when the set is empty. Therefore this mutant is killed by all test suites.

```
COI_20:115:OrdSetSimple_difference(OrdSetSimple): s2.binSearch(
s1.getElementAt( k ) ) < 0  =>  !(s2.binSearch( s1.getElementAt( k ) ) <
0)
```

This will cause all items in first set to be added to the test suite. Unless the first set is empty, this mutant is killed. Since all test suites contain a test case where the difference set is not empty, this mutant is killed.

```
COI_21:126:java.lang.String_toString(): k <= _last  =>  !(k <= _last)
   for (k = 0; !( k <= _last); k++)
```

When the set is empty, the program goes into this part of the code. Since the value of last is -1 and the set is empty, accessing the set at 0 will cause the program to return an array out of bound exception. Since all test suites contain a test case where the difference set is empty, this mutant is killed by all of them.

```
COI_22:134:boolean_equals(java.lang.Object): this == obj  =>  !(this ==
obj)
COI_23:140:boolean_equals(java.lang.Object): this == null  =>  !(this ==
null)
COI_24:140:boolean_equals(java.lang.Object): null == obj  =>  !(null ==
obj)
COI_25:140:boolean_equals(java.lang.Object): this == null || null == obj
=>  !(this == null || null == obj)
COI_26:145:boolean_equals(java.lang.Object): this.getSize() == 0  =>
!(this.getSize() == 0)
COI_27:145:boolean_equals(java.lang.Object): ph.getSize() == 0  =>
!(ph.getSize() == 0)
COI_28:145:boolean_equals(java.lang.Object): this.getSize() == 0 &&
ph.getSize() == 0  =>  !(this.getSize() == 0 && ph.getSize() == 0)
COI_29:148:boolean_equals(java.lang.Object): i < this.getSize()  =>  !(i <
this.getSize())
```

```
COI_30:149:boolean_equals(java.lang.Object): this.getElementAt( i ) !=
ph.getElementAt( i )  =>  !(this.getElementAt( i ) != ph.getElementAt( i
))
COI_31:154:boolean_equals(java.lang.Object): this.getSize() ==
ph.getSize()  =>  !(this.getSize() == ph.getSize())
COI_33:158:boolean_equals(java.lang.Object): toreturn  =>  !toreturn
```

These mutants are not killed by any of the test suites because the equals method is never used.

```
LOI_1:16:OrdSetSimple(int):size => ~size
```

This change will set the set's size to a negative value when it is positive therefore this mutant is killed by all test suites.

```
LOI_2:17:OrdSetSimple(int):_set_size => ~_set_size
```

This mutant is killed because the value of set_size is changed to a negative value (when it's positive).

```
LOI_3:18:OrdSetSimple(int):k => ~k
LOI_4:18:OrdSetSimple(int):_set_size => ~_set_size
LOI_5:18:OrdSetSimple(int):k => ~k
```

LOI_3 is killed by all mutants. Since changing the value will cause k to be always less than zero, the loop does not stop until there is an array out of bounds condition because of setting a value that does not exist.

```
LOI_6:19:OrdSetSimple(int):k => ~k
```

This mutant is killed by all test suites because changing the value of k to ~k will cause the value to be negative causing an array out of bounds exception.

```
LOI_7:29:int_make_a_free_slot(int):val => ~val
```

Since there are no duplicates any of the sets, the program will not be able to find val since it does not exist in the set. Therefore this change will have no effect. This mutant is not killed by any of the test suites.

```
LOI_8:30:int_make_a_free_slot(int):pos => ~pos
```

This will cause the program to only add values to position 1 in the set. This mutant is killed by all test suites because the resulting difference set will only contain 2 items at most.

```
LOI_9:33:int_make_a_free_slot(int):i => ~i
for (i = 0; i <= _last; i++) {
```

This mutant is killed by all test suites. ~0 will be equal to -1 which is equal to last, this will cause the program to compare this to last and stay in the loop. Eventually the program cannot access any more items in the list and returns an array out of bound exception. Since all test suites contain a test case where the set is not empty, this mutant is killed by all test suites.

LOI_10:33:int_make_a_free_slot(int):_last => ~_last

This change causes the system to enter this loop and change the value of i to 1 leaving the first item at 0. By changing the value of i, the system will enter the loop that is followed by this statement causing the set to be in reverse order. When in reverse order the program will have an incorrect difference set. Therefore this mutant is killed by all test suites.

```
LOI_12:34:int_make_a_free_slot(int):i => ~i
```

This change causes the program to try to access a negative value for the index which results in an array out of bounds exception. Since all test suites contain a set that is non-empty this mutant is killed by all.

```
LOI_13:34:int_make_a_free_slot(int):val => ~val
```

This change will cause the program to make the set into reverse order. This change will cause the difference set to return zero. Therefore this mutant is killed by all test suites.

```
LOI_14:38:int_make_a_free_slot(int):i => ~i
LOI_15:38:int_make_a_free_slot(int):_last => ~_last
LOI_16:38:int_make_a_free_slot(int):_last => ~_last
LOI_17:39:int_make_a_free_slot(int):_last => ~_last
LOI_18:39:int_make_a_free_slot(int):j => ~j
LOI_19:39:int_make_a_free_slot(int):i => ~i
```

LOI_14 is not killed by any of the test suites. If the value of i is changed and the program enters the loop the position will be set to i which will not make a change to the functionality. Therefore this mutant is not killed because it is equivalent

LOI_15 and LOI_17 are killed by the Each Choice test suite because it contains a test case where the elements are not ordered. This change will prevent the program from entering this loop thereby not ordering the set. When compared to the ordered set the program will return an empty difference set. For an ordered set items will be added to the end of the set and this change does not have any effect.

LOI_16 is not killed by any of the test suites. If the value of last is -1 and the program does not enter the loop, the position is set to _last +1 which is 0 and the same as i. Therefore this mutant is equivalent and not killed by any of the test suites.

LOI_18 and LOI_19 are killed by all test suites. Since all test suites cover this part of the code and changing the operator will cause the program to enter/ not enter this loop, all test suites are able to kill these mutants.

```
LOI_21:40:int_make_a_free_slot(int):j => ~j
LOI_22:40:int_make_a_free_slot(int):j => ~j
```

These two mutants will only have an effect if the set is not ordered. Therefore they are only killed by the Each Choice test suites since this is the only test suite that has a test case with an unordered set.

```
LOI_23:42:int_make_a_free_slot(int):i => ~i
```

This mutant is killed by all test suites because of the same reason as AOIS_74.

```
LOI_24:44:int_make_a_free_slot(int):_last => ~_last
```

This mutant is killed by all test suites. This change causes the position to be always 0. This is because when the set is empty i.e. last is -1, the value of _last changes to 0. The next time we enter the loop the new value of last is 0 and ~_last+1 is equal to 0 again. Since all test suites contain non empty value for the difference set, this mutant is killed by all test suites.

```
LOI_25:46:int_make_a_free_slot(int):_last => ~_last
```

This addition causes a syntax error and cannot be complied therefore it is killed by all test suites.

```
LOI_26:47:int_make_a_free_slot(int):pos => ~pos
```

This will cause the position to be negative and causes to values to be added to the sets. Since all test suites contain a test case with non-empty values, this mutant is killed by all test suites.

```
LOI_27:53:void_addElement(int):n => ~n
```

This will cause no elements to be added to the sets. Since the difference set will also be empty, when compared to the actual difference set the program returns false therefore these mutants will be killed by all test suites.

```
LOI_28:57:void_addElement(int):_last => ~_last
```

We will enter this loop if the user is trying to add a value when the set is already reached its maximum. By changing last+1 to ~_last+1, if the size of the set is 1, the value of ~_last+1 will be 0+1 which is 1, and therefore the value cannot be added. Since no test suite contains a test case where one of the sets contain only one value, this mutant is not killed.

```
LOI_29:57:void_addElement(int):_set_size => ~_set_size
```

LOI_29 is not killed by any of the test suites. This will cause no items to be added to the test suite because the array size will be set to full. Since the difference set is also empty, when compared to the actual difference set, the program returns false. Therefore this mutant is killed by all test suites.

```
LOI_30:58:void_addElement(int):n => ~n
```

This mutant does not make a difference in the functionality since it is in the print statement. Therefore it is not killed by any of the test suites.

```
LOI_31:62:void_addElement(int):n => ~n
LOI_32:63:void_addElement(int):pos => ~pos
LOI_33:64:void_addElement(int):pos => ~pos
LOI_34:64:void_addElement(int):n => ~n
```

These mutants are killed by all test suites. LOI_31 is killed by all test suites and this is because changing this value causes the program to re-order the sets. Since the elements of the set are not ordered, when compared to the correct difference set, the program returns the first set. Therefore it is killed by all test suites.

```
LOI_35:72:int_getSize():_last => ~_last
```

This mutant is killed by all test suites. This change will cause the program to return a negative value for the set when the set is not empty. Therefore this will cause an index out of bound exception and is killed by all mutants.

```
LOI_36:77:int_getElementAt(int):i => ~i
LOI_37:77:int_getElementAt(int):i => ~i
LOI_38:77:int_getElementAt(int):_last => ~_last
```

LOI_36 causes the program to return a -1 value since ~i will always be negative. This causes the program to try and add a negative value to the set which causes the set to be empty. Therefore this mutant is killed by all test suites.

LOI_37 is not killed because there are no test cases where the value of i is greater than the set's size. This is because this method is not being tested.

LOI_38 are killed by all test suites. This change causes the program to have a negative value for last which makes it less than i and the program returns -1. The program cannot add -1 since it is a negative value and therefore when comparing to the actual difference set, the program returns false. Therefore this mutant is killed by all test suites.

```
LOI_39:80:int_getElementAt(int):i => ~i
```

This mutant is killed by all test suites. The program tries to access a negative value after this change is made. Therefore the method returns an index out of bound exception and the mutants are killed.

```
LOI_40:87:int_binSearch(int):_last => ~_last
```

This mutant is killed by all test suites. This change causes j to always be less than i and the set to return the value of i, meaning all values of first set will be added to the difference set. Therefore this mutant is killed by all test suites.

```
LOI_41:89:int_binSearch(int):_last => ~_last
```

This mutant is killed by all test suites. When the second set is empty this expression will evaluate to false and the program will try to access an item on the set that is non-existent since the set is empty. This causes an exception and the mutant is therefore killed by all test suites.

```
LOI_42:92:int_binSearch(int):i => ~i
LOI_43:92:int_binSearch(int):j => ~j
```

These mutants are killed by all test suites. Changing these values will cause the program to go into an infinite loop since one value will never reach the other because one is always negative.

```
LOI_44:93:int_binSearch(int):i => ~i
LOI_45:93:int_binSearch(int):j => ~j
```

These mutants are killed by all test suites. This change causes the program to go to an infinite loop because when the program reaches the half point in the binary search, the value of m will stay the same and the program gets stuck in a infinite loop.

```
LOI_46:94:int_binSearch(int):x => ~x
LOI_47:94:int_binSearch(int):m => ~m
LOI_48:95:int_binSearch(int):m => ~m
LOI_49:97:int_binSearch(int):m => ~m
LOI_50:100:int_binSearch(int):x => ~x
LOI_51:100:int_binSearch(int):i => ~i
LOI_52:101:int_binSearch(int):i => ~i
```

LOI_46 to LOI_52 will cause the program to be in a state of an infinite loop. Therefore these mutants are all killed by all test suites.

```
LOI_53:112:OrdSetSimple_difference(OrdSetSimple):size1 => ~size1
```

This mutant is killed by all test suites because it sets the size of the set to a negative value and this will return an array out of bound exception.

```
LOI_54:114:OrdSetSimple_difference(OrdSetSimple):k => ~k
LOI_55:114:OrdSetSimple_difference(OrdSetSimple):size1 => ~size1
LOI_56:114:OrdSetSimple_difference(OrdSetSimple):k => ~k
```

LOI_54 and LOI_65 will cause syntax errors and are therefore killed by all test suites.

LOI_55 changes the size to ~size1 which causes to values to be added to the difference set because k is greater than ~size.

```
LOI_57:115:OrdSetSimple_difference(OrdSetSimple):k => ~k
```

This mutant is killed by all test suites. This mutant will cause the program to try to get the element from a negative value which does not exist. Since all test suites contain a choice that contains non-empty sets, this mutant is killed by all test suites.

```
LOI_58:116:OrdSetSimple_difference(OrdSetSimple):k => ~k
```

This mutant is killed because of the same reason as LOI_57.

```
LOI_59:126:java.lang.String_toString():k => ~k
LOI_60:126:java.lang.String_toString():_last => ~_last
LOI_61:126:java.lang.String_toString():k => ~k
```

These mutants cause the program not to enter the loop and no values to be added to the strings to compare. Therefore these mutants are killed by all test suites.

```
LOI_62:127:java.lang.String_toString():k => ~k
```

This mutant is killed by all test suites because of the same reason as LOI_59 to LOI_61.

```
LOI_63:148:boolean_equals(java.lang.Object):i => ~i
LOI_64:148:boolean_equals(java.lang.Object):i => ~i
LOI_65:149:boolean_equals(java.lang.Object):i => ~i
LOI_66:149:boolean_equals(java.lang.Object):i => ~i
```

These mutants are not killed because the equlas method is never used by any of the test suites.

APPENDIX F TAXI BILLING CASE STUDY

F1. CP Specification

Parameter	Category	Choice	Property	constraint
Destination	**[Cat1]**Area of destination	**[Ch1.1]** Urban destination		[Base Choice]
		[Ch1.2] Suburban destination		
		[Ch1.3]Beyond Suburban destination		
		[Ch1.4] Destination not informed or unknown		[error]
departureTime	**[Cat2]** Time of departure	**[Ch2.1]**Negative time		
		[Ch2.2] Night		
		[Ch2.3] Morning		[if !Sunday]
		[Ch2.4] Afternoon		[if !Sunday][Base Choice]
		[Ch2.5] Early evening		[if !Sunday]
		[Ch2.6] Late evening		[if !Sunday]
		[Ch2.7] Illegal time		[error]
		[Ch2.8] Sundays special time of departure		[if !Weekday]
dayOfWeek	**[Cat3]** Day of the week	**[Ch3.1]** Day not informed or unknown		[error]
		[Ch3.2] Weekdays and Saturdays	[Weekday]	[Base Choice]
		[Ch3.3] Sundays	[Sunday]	
holiday	**[Cat4]** Holiday	**[Ch4.1]** It is a holiday True		
		[Ch4.2] It is not a holiday false		[Base Choice]
kmRun	**[Cat5]** Kilometres run	**[Ch5.1]** Illegal distance run		[error]
		[Ch5.2] Legal distance run		[Base Choice]

Table 60 TaxiBilling Case Study: CP Specification

F2. Test Frames

#	Test Frame	Test Case	Output
1	C 1.1 |C 2.4 |C 3.2 | C 4.2 |C 5.2 |	"Paris", 16, "Monday", false, 3.80	totalCost: 5.8 charge: A
2	C 1.2 |C 2.4 |C 3.2 | C 4.2 | C 5.2 |	"End of area", 11, "Tuesday", false, 5.3	totalCost: 8.136 charge: B
3	C 1.3 |C 2.4 |C 3.2 | C 4.2 |C 5.2 |	"Beyond suburban", 15, "Wednesday", false, 11.2	totalCost: 17.32 charge: C
4	C 1.1 |C 2.2 |C 3.2 | C 4.2 |C 5.2 |	"Paris", 0, "Thursday", false, 3.42	totalCost: 6.0304 charge: B
5	C 1.1 |C 2.3 |C 3.2 |	"Paris", 8, "Friday", false, 2	totalCost: 5.8

	C 4.2 |C 5.2 |		charge: B
6	C 1.1 |C 2.5 |C 3.2 | C 4.2 |C 5.2 |	"Paris", 18, Saturday", false, 9.71	totalCost: 13.0752 charge: B
7	C 1.1 |C 2.6 |C 3.2 | C 4.2 |C 5.2 |	"Paris", 20, "Friday", false, 31.6	totalCost: 37.592 charge: B
8	C 1.1 |C 2.4 |C 3.2 | C 4.1 |C 5.2 |	"Peripheral road", 11, "Monday", true, 14.0	totalCost: 17.88 charge: B
9	C 1.4 |	"", 10 , "Monday", false, 10.0	totalCost:-1.0 charge:
10	C 2.1 |	"Paris",-2, "Monday", false, 10	totalCost: -1.0 charge:
11	C 2.7 |	"Paris", 25, "Monday", false, 10	totalCost:-1.0 charge:
12	C 3.1 |	"Paris", 10, " ", false, 10	totalCost: charge:
13	C 5.1 |	"Paris", 10, "Wednesday", **false**, -1	totalCost:-1.0 charge:

Table 61 TaxiBilling Case Study: Base Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.6 |C 3.2| C 4.1 |C 5.2 |	"End of area", 21, "Monday", true, 12.35	totalCost: 18.8725 charge: C
2	C 1.1 |C 2.4 |C 3.2| C 4.2 |C 5.2 |	"Paris", 11, "Tuesday", false, 12.35	totalCost:12.821 charge: A
3	C 1.3 |C 2.5 |C 3.2| C 4.1 |C 5.2 |	"Beyond suburban", 18, "Wednesday", true, 5	totalCost: 8.95 charge: C
4	C 1.3 |C 2.3 |C 3.2| C 4.2 |C 5.2 |	"Beyond suburban", 9, "Thursday", false, 10.5	Total Cost: 16.375 charge: C
5	C 1.1 |C 2.8 |C 3.3| C 4.1 |C 5.2 |	"Peripheral road", 7, "Sunday", true, 9.2	Total Cost: 12.504 charge: B
6	C 1.1 |C 2.2 |C 3.2| C 4.2 |C 5.2 |	"Paris", 5, "Wednesday", **false**, 5.5	Total Cost: 8.36 charge: B
7	C 1.4 |	"", 10, "Friday", **false**, 10.0	totalCost:-1.0 charge:
8	C 2.1 |	"Paris", -2, "Saturday", **false**, 10.0	totalCost:-1.0 charge:
9	C 2.7 |	"Paris", 25, "Monday", **false**, 10.0	totalCost:-1.0 charge:
10	C 3.1 |	"Paris", 10, "", **false**, 10.0	totalCost:-1.0 charge:
11	C 5.1 |	"Paris", 10, "Wednesday", **false**, -1	totalCost:-1.0 charge:

Table 62 TaxiBilling Case Study: Each Choice Test Frames

#	Test Frame	Test Case	Output
1	C 1.2 |C 2.3 |C 3.2| C 4.1 |C 5.2 |	"End of area", 7, "Monday", **true**,4	totalCost: 7.6 Charge: C
2	C 1.2 |C 2.6 |C 3.2 | C 4.2 |C 5.2 |	"Orly", 19, "Tuesday", **false**, 14.5	totalCost: 21.775 Charge: C
3	C 1.3 |C 2.5 |C 3.2 | C 4.1 |C 5.2 |	"Beyond suburban", 18, "Wednesday", **true**, 21.5	totalCost: 31.225 Charge: C
4	C 1.1 |C 2.2 |C 3.3 | C 4.2 |C 5.2 |	"Paris", 5, "Sunday", **false**, 13	totalCost: 19.75 Charge: C
5	C 1.2 |C 2.2 |C 3.2 | C 4.1 |C 5.2 |	"Roissy", 0, "Thursday", **true**, 14	Total Cost: 21.1 Charge: C
6	C 1.2 |C 2.8 |C 3.3 |	"Du Parc", 7, "Sunday", **true**, 18.5	totalCost: 27.175

	C 4.1 |C 5.2 |		Charge: C
7	C 1.1 |C 2.8 |C 3.3 | C 4.2 |C 5.2 |	"Peripheral road",23, "Sunday", **false**, 12	totalCost: 15.64 Charge: B
8	C 1.3 |C 2.4 |C 3.2 | C 4.2 |C 5.2 |	"Beyond suburban", 15, "Friday", **false**, 3	totalCost: 6.25 Charge: C
9	C 1.1 |C 2.4 |C 3.2 | C 4.1 |C 5.2 |	"Paris", 10, "Sunday", **true**, 34.4	totalCost: 40.728 Charge: B
10	C 1.2 |C 2.5 |C 3.2 | C 4.2 |C 5.2 |	"Villepinte", 18, "Saturday",**false**, 32	totalCost: 38.04 Charge: B
11	C 1.1 |C 2.6 |C 3.2| C 4.1 |C 5.2 |	"Peripheral road", 19, "Monday", **true**, 8	totalCost: 11.16 Charge: B
12	C 1.2 |C 2.4 |C 3.2 | C 4.1 |C 5.2 |	"End of area", 10, "Tuesday", **true**, 42	totalCost: 58.9 Charge: C
13	C 1.3 |C 2.6 |C 3.2 | C 4.1 |C 5.2 |	"Beyond suburban", 7, "Wednesday",**true**, 15.2	totalCost: 22.72 Charge: C
14	C 1.1 |C 2.3 |C 3.2 | C 4.2 |C 5.2 |	"Paris", 9, "Monday", **false**, 11	totalCost: 14.52 Charge: B
15	C 1.3 |C 2.3 |C 3.2 | C 4.2 |C 5.2 |	"Beyond suburban", 9, "Wednesday", **false**, 9	totalCost: 14.35 Charge: C
16	C 1.3 |C 2.2 |C 3.3 | C 4.1 |C 5.2 |	"Beyond suburban", 3, "Sunday", **true**, 25	totalCost: 35.95 Charge: C
17	C 1.1 |C 2.5 |C 3.2 | C 4.2 |C 5.2 |	"Peripheral road", 17, "Thursday", **false**, 13	totalCost: 16.76 Charge: B
18	C 1.3 |C 2.8 |C 3.3 | C 4.1 |C 5.2 |	"Beyond suburban", 22, "Sunday", **true**, 42	totalCost: 58.9 Charge: C
19	C 1.4 |	"", 7, "Sunday", **false**, 18.5	totalCost: -1 Charge:
20	C 2.1 |	"Paris", -2, "Monday", **false**, 10.0	totalCost: -1 Charge:
21	C 2.7 |	"Paris", 25, "Monday", **false**, 10.0	totalCost: -1 Charge:
22	C 3.1 |	"Paris", 10, "", **false**, 10.0	totalCost: -1 Charge:
23	C 5.1 |	"Paris", 10, "Wednesday", **false**, -1	totalCost: -1 Charge:

Table 63 TaxiBilling Case Study: Pair Wise Test Frames

F3. Explanation of Mutants Killed or Alive

a. TaxiBilling Class:

JSI_1 to JSI_6 change the following variables to static:

```
private TaxiDao dao;
private String destinationArea;
private int departureTime;
private WeekDay day;
private boolean holiday;
private BigDecimal totalKm;
```

Since the values of these variables are not changed anywhere in the program, changing the type to static will not have an effect on the functionality. These mutants are therefore equivalent and not killed by any of the test suites.

JTD_3 to JTD_7 remove `this` keyword from the left hand side of the following statements in the second constructor, while JTI_6 to JTI_10 insert `this` keyword on the right hand side.

```
this.destinationArea = destinationArea;
this.departureTime = departureTime;
this.day = day;
this.holiday = holiday;
this.totalKm = totalKm;
```

The initial values of `destinationArea` and `departureTime` are `null`. By removing the `this` keyword the value of `destinationArea` and `departureTime` are set to their own value causing the value of the object's variables to be `null`. Since all test cases use this constructor to set the values, JTD_3 and JTD_4 are killed by all test suites.

Adding a `this` keyword to the right hand side of the first and second line in the statement will cause the object's variables to be set to `null`. Therefore JTI_6 and JTI_7 are also killed by all test suites.

JTD_5 is not killed by any of the test suites. This is because the first constructor uses the second constructor to set day to `null`. Since this constructor is never used by any of the test suites to set the value of date, this mutant is not killed. JTD_7 is not killed because of the same reason, but with the value of `totalKm is` set to `null` instead of day. JTI_8 and JTI_10 are not killed because of the same reason as JTD_5 and JTD_7.

JTD_6 causes the value of `holiday` to be set to it and leave the value of the object's holiday to its initial value which is `false`. The test cases where the value of `holiday` is `false` do not make any contribution to killing this mutant. The mutant will only be killed if there is a test case where changing the value of `holiday` from false to true will change the applicable charge. The Base Choice and the Pair Wise test suites contain a choice where changing holiday will have an impact on the `applicableCharge`. The Each Choice test suite however does not contain such a test case and is not able to kill this mutant. This is the same case with JTI_9 with the difference that this object's holiday is set to its own value which is false.

PRV_1 changes `totalCost = minimumCharge` in

```
if (totalCost.compareTo( minimumCharge ) < 0) {
totalCost = minimumCharge;
```

to `totalCost = totalKm;`

This mutant will be killed if the value of `totalCost` is less than the minimum charge. Since there are no test cases with minimum charge in the Each Choice and the Pair Wise test suites this mutant is not killed by these two. The Base Choice test suite however contains such test case and is able to kill this mutant.

PRV_2 changes `this.dao = taxiDao` to `this.dao = dao.`
Since the constructor for a test case uses this function and this change causes the value to be set to null, this mutant is killed by all test suites.

AODU_1 changes `return new melba.taxibilling.CalculationResult( new java.math.BigDecimal( -1 ), "" )` to `return new CalculationResult(new BigDecimal(1), "");`
Since all test suites have a test case where there are invalid values for day and total kilometer, the program tries to compare the value returned in reality which is -1 to 1 and returns false. Therefore this mutant is killed by all test suites.

AOIS_11 and AOIS_12 change the following line of code:

```
ApplicableCharge applicableCharge = dao.findApplicableCharge( day,
holiday, departureTime++, destinationArea);
```

The value of `departureTime` is never used after this statement. Therefore it is equivalent and not killed by any of the test suites.

AOIU_4 changes `the value of departureTime` in `ApplicableCharge applicableCharge = dao.findApplicableCharge( day, holiday, departureTime, destinationArea )`
to `--departureTime.`
A negative value will cause the `applicableCharge` to be -1. Since all test suites have test cases that use this function to find the applicable charge, this mutant is killed by all test suites.

COI_4 changes `if (day != null)` to `if (!(day != null)).`
Since all test suites contain a test case with a valid day, this mutant is killed by all test suites.

COI_5 Changes `if (totalKm.compareTo( new java.math.BigDecimal( 0 ) ) == 1)` to `if (!(totalKm.compareTo( new java.math.BigDecimal( 0 ) ) == 1)).`

If the input of `totalKm` is greater than 0, this part of the `if` statement evaluates to false and the program sets the total charge to -1. Since all test suites contain test cases where the `totalKm` is greater than 0, this mutant is killed by all test suites.

COI_6 changes `melba.taxibilling.model.ApplicableCharge applicableCharge = dao.findApplicableCharge( day, holiday, departureTime, destinationArea );` to `melba.taxibilling.model.ApplicableCharge applicableCharge = dao.findApplicableCharge( day, !holiday, departureTime, destinationArea );`
If the test suite contains a test case where changing the value of holiday from true to false or vice versa causes the applicable charge to be different than the actual value, this mutant is killed. Since all test suites contain such a test case, the mutant is killed by all test suites.

COI_7 changes `if (applicableCharge != null)` to `if (!(applicableCharge != null))`.
All test suites contain a choice where the value of applicable charge is not null. If this value is not null, the mutant evaluates to false and the program does not calculate the right amount for the `totalCost`. Therefore this mutant is killed by all test suites.

COI_8 changes `if (totalCost.compareTo( minimumCharge ) < 0)` to `if (!(totalCost.compareTo( minimumCharge ) < 0)) {`

This change causes the program to set the totalCost to minimum charge for all test cases that have a `totalCost` greater than minimum charge. Since all test suites contain a test case with totalCost greater than `minimumCharge`, this mutant is killed by all test suites.

LOI_4 changes `melba.taxibilling.model.ApplicableCharge applicableCharge = dao.findApplicableCharge( day, holiday, departureTime, destinationArea );` to `melba.taxibilling.model.ApplicableCharge applicableCharge = dao.findApplicableCharge( day, holiday, ~departureTime, destinationArea )`.

This mutant changes the value of `departureTime` to negative if it is positive and vice versa. If the value is negative and it changes to positive because of this change, the `totalCost` will no longer be equal to -1. The opposite applies if the value is positive and changed to negative. The only case where this mutant is not killed is when the time is an invalid value of greater than 23.

This change causes the value to be changed to a negative value which is still invalid and the result is the same. Since all test cases have valid and invalid values of less than 1, this mutant is killed by all.

ROR_1 `changes if (day != null)` to `if (day == null) {`

This mutant is the same as COI_4 and is therefore killed by all test suites.

ROR_2 changes `if (totalKm.compareTo( new java.math.BigDecimal( 0 ) ) == 1)` to `if (totalKm.compareTo( new java.math.BigDecimal( 0 ) ) > 1).`

The return value of the statement will never be greater than 1, since `compareTo` method returns -1,0 or 1, when the first value is less than, equal to or greater than the second value. This causes the program to not enter this loop if the value of `totalKm` is greater than 0. Since all test suites contain test cases where the value of `totalKm` is greater than 0, this mutant is killed by all test suites.

ROR_3 changes `if (totalKm.compareTo( new java.math.BigDecimal( 0 ) ) == 1)` to `if (totalKm.compareTo( new java.math.BigDecimal( 0 ) ) >= 1) {`

The value returned by the `compareTo` method will never be greater than 1. Therefore this mutant is equivalent and not killed by any of the test suites.

ROR_4 to ROR_7 change the equals operator in `(totalKm.compareTo( new java.math.BigDecimal( 0 ) ) == 1)` to <, <= and !=.

All test suites contain choices where the value of `totalKm` is greater than zero and less than zero. This change causes the program to return -1 when we have a valid `totalKm`. Therefore these mutants are killed by all test suites.

AOIS_1 and AOIS_2 change departureTime in this(destinationArea, departureTime, null, holiday, null) to departureTime++ and departureTime--.

Since the value of `departureTime` is never used after this statement, these mutants are equivalent and not killed by any of the test suites.

AOIS_3 and AOIS_4 change `totalKm in this.totalKm = new java.math.BigDecimal( totalKm )` to `++totalKm and --totalKm`.

If the value of `totalKm` is increased by 1 it will change the `totalCost` and therefore is caught by all test cases that have valid inputs. If the value of `totalKm` is decreased it is killed by all test cases that contain a valid `totalKm` except for the value of `totalKm` that leads to having a minimum charge. If the charge's value is minimum and it is greater than 2 the `totalCost` will still remain the same. All mutants are able to kill these mutants since they contain test cases with valid inputs that result in `totalCost` greater than the minimum charge.

AOIS_5 and AOIS_6 change `totalKm` in `this.totalKm = new java.math.BigDecimal( totalKm )` to `totalKm++ and totalKm--`. Since `totalKm` is never used after this statement. These two mutants are not killed by any of the test suites since they are equivalent.

AOIU_1 changes `this( destinationArea, departureTime, null, holiday, null )` to `this( destinationArea, -departureTime, null, holiday, null )`.

If the program has a valid `departureTime` the value will be turned into negative and cause the `totalCost` to be -1 where it should be a positive value. If the program has an invalid value that is less than 0 the program will return a positive value and therefore the program will return a `totalCost` that is greater than 0. If the program has an invalid value greater than 23, the mutant is not caught because the value will still be invalid. Therefore this mutant is killed by all test suites.

AOIU_2 changes `this.totalKm = new BigDecimal( totalKm )` to `this.totalKm = new BigDecimal( -totalKm )`. Given a valid `totalKm` will cause the program to return -1 for the `totalCost` which is not correct and to return a `totalCost` when the value of `totalKm` is negative. This mutant is killed by any test case that has valid or invalid values or `totalKm`.

COI_1 changes `this( destinationArea, departureTime, null, holiday, null );` to `this( destinationArea, departureTime, null, !holiday, null )`.

This change reverses the value of `holiday`. If there is a test case that changes the `applicableCharge` when the holiday value is changed, then this mutant is killed. Please refer to Table 3for further details on applicable charge. Since all test suites contain such a test case, this mutant is killed by all test suites.

LOI_1 changes `this( destinationArea, departureTime, null, holiday, null )` to `this( destinationArea, ~departureTime, null, holiday, null )`

This mutant is killed by all test suites because of the same reason as AOIU_1.

AOIS_9 and AOIS_10 change `departureTime` in `this.departureTime = departureTime` to `departureTime++` and `departureTime--`. Since the value is never used after this statement, these mutants are equivalent and not killed by any of the test suites.

AOIS_7and AOIS_8 change `this.departureTime = departureTime` to `this.departureTime = ++departureTime` and `this.departureTime = --departureTime`.

If the test cases contain border values and a change in the value causes the program to return a different `ApplicableCharge`, these mutants are caught. Since all test suites contain such test cases these mutants are killed by all test suites.

AOIU_3 changes `this.departureTime = departureTime` to `this.departureTime = -departureTime`.

This change causes the program to return -1 if there is a valid `departureTime` and return an actual value when there is an invalid `departureTime`. Since all test suites contain test cases with valid and invalid `departureTime`, this mutant is killed by all test suites.

COI_3 changes `this.holiday = holiday;` to `this.holiday = !holiday;`

This mutant is killed because of the same reason provided for COI_1.

LOI_3 changes `this.departureTime = departureTime` to `this.departureTime = -departureTime`.

This mutant is killed because of the same reason as AOIU_3.

b. TaxiDaoStructure Class:

JDC_1 removes the constructor from the code. This change causes the value of `taxiDaoStructure` to not be set and is therefore killed by all test suites.

JSD_1 to JSD_5 remove the static keyword from the following lines of code:

```
private static java.math.BigDecimal INITIAL_CHARGE;
private static java.math.BigDecimal MINIMUM_CHARGE;
private static melba.taxibilling.model.DestinationAreaType urban;
private static melba.taxibilling.model.DestinationAreaType suburb;
private static melba.taxibilling.model.DestinationAreaType beyondSuburban;
```

These mutants remain alive because these values are never changed throughout the program. These mutants are equivalent.

JSI_1 to JSI_3 add static keyword to the following lines of code:

```
Charge a;
Charge b;
Charge c;
```

Since these values are never changed throughout the program, changing the value to `static` will not have an impact on the functionality. Therefore these mutants are not killed by any of the test suites because they are equivalent.

AOIS_10 changes departureTime in

```
if (day == WeekDay.SUNDAY) {
if{
…
}
else if (departureTime >= 7 && departureTime < 24) {
…
}
```

to `--departureTime >= 7`

There are no test cases in the Base Choice test suite where the value of day is equal to `Sunday`. Therefore this test suite is not able to kill this mutant. The Each Choice test suite contains a test case with the value of day equal to `Sunday` and it does reach this part of the code, however the value `departureTime` for this particular test case is not at the border which is `>=7`. If the value of `departureTime` is changed to 7, this mutant will be killed. The Pair Wise test suite contains a test case with the above mentioned description and it is able to kill this mutant.

AOIS_1 changes the same part of the code as AOIS_10 to `++departureTime >= 7.` If the value of day is `Sunday` and the `departureTime` is 23, this mutant is killed because it changes the value of `departureTime` to 24 which will result in an invalid value in the next check. Since the Pair Wise test suite contains such a test case it is able to kill this mutant. The Each Choice test suite also contains a test case for `Sunday` but the value of the `departureTime` is not 23, therefore it is not able to kill this mutant. The Base Choice test suite does not contain a test case with `Sunday` and is therefore unable to kill this mutant.

AOIS_2 changes the same part of the code as AOIS_10 to `--departureTime >= 7.` If the value of day is `Sunday` and the `departureTime` is 7, the `if` statement will return true instead of false and the `applicableCharge` will change. The Each Choice test suite contains such a test case but the other two test suites do not contain such a test case. Therefore the Each Choice test suite kills this mutant while the Base Choice and Pair Wise test suites are not able to kill this mutant.

AOIS_3 changes the same part of the code as AOIS_10 to `departureTime++ >= 0.` This mutant is killed by the Pair Wise test suite but not by the Each Choice and the Pair Wise test suites. This is because the Pair Wise test suite contains a test case where the value of `departureTime` is 23. After the execution of this statement the `departureTime` is increased by one and the value becomes 24. This statement will return false instead of true and therefore this mutant is killed. The Each Choice and the Base Choice test suites do not contain such a test case and cannot kill this mutant.

AOIS_4 changes the same part of the code as AOIS_10 to `departureTime-- >= 0.` If there is a test case where the value of day is `Sunday` and the time is 0 and changing the time has an impact on the `applicableCharge`, this mutant is killed. No test suite contains such a test case therefore this mutant is not killed by any of the test suites.

AOIS_5 changes the same part of the code as AOIS_10 to `++departureTime < 7.` This mutant is killed by the Pair Wise test suite. This test suite contains a choice where the value of day is `Sunday` and the `departureTime` is 23. After this statement is executed the value of `departureTime` is increased by one and the next statement will return false. Therefore this

mutant is killed. The Each Choice and the Base Choice test suites do not have such a test case and are not able to kill this mutant.

AOIS_6 changes the same part of the code as AOIS_10 to `--departureTime < 7.`

This mutant is killed by the Each Choice test suite since it contains a choice with `departureTime` equal to 7. This will cause this statement to return true even though it is false. The other test suites do not have such a test case and therefore are unable to kill this mutant.

AOIS_7 changes the same part of the code as AOIS_10 to `departureTime++ < 7.`

AOIS_8 changes the same part of the code as AOIS_10 to `departureTime--< 7.`

AOIS_9 changes the same part of the code as AOIS_10 to `++departureTime >= 7.`

AOIS_7 to AOIS_9 are killed by the Pair Wise test suite because of the same logic as AOIS_1 to AOIS_5.AOIS_11 changes the same part of the code as AOIS_10 to `departureTime++ >= 7.` This time the impact will be on the statement followed after this statement. Therefore if there is a test case that has day equal to `Sunday` and `departureTime` equal to 23, this mutant will be killed. The Pair Wise test suite contains such a test case and is therefore able to kill this mutant. The Base Choice test suite does not contain a test case with `Sunday` and is not able to kill this mutant. The Each Choice test suite only contains one test case with this criterion and it is not able to kill this mutant. If the value of the `departureTime` is set to 23 then the Each Choice test suite will also be able to kill this mutant.

AOIS_12 changes the same part of the code as AOIS_10 to `departureTime-->=7.` This change will have no impact on the return statement. The program will evaluate this statement and decrease the value of `departureTime` for the next statement which is `departureTime<24` which evaluates to true. Therefore this mutant is equivalent and not killed by any of the test suites.

AOIS_13 changes the same part of the code as AOIS_10 to `++departureTime < 24.`

If the test suite contains a test case with value of day equal to `Sunday` and the `departureTime` equal to 23, this mutant will be killed. This is because a value of 23 will make this statement return false when it should return true. Since the Pair Wise test suite contains such a test case, this mutant is killed. The Each Choice has a test case for `Sunday` but the value of the `departureTime` is not 23 and therefore this test suite is not able to kill this mutant. The Base Choice test suite does not contain a test case for `Sunday` and is therefore unable to kill this mutant.

AOIS_14 changes the same part of the code as AOIS_10 to `--departureTime < 24.` This mutant will be killed if the value of the day is `Sunday` and the value of the `departureTime` is 24. The mutant will make the program return true even though the value of 24 is invalid. Since there are no test cases in any of the test suites that have such a test case, this mutant is not killed by any of the test suites.

AOIS_15 and AOIS_16 change the same part of the code as AOIS_10 to `departureTime++ < 24` and `departureTime-- < 24.` Since the value of `departureTime` is never used after this statement, these mutants are equivalent and therefore not killed by any of the test suites.

AOIS_17 changes departureTime>=0 in if (holiday) { if (departureTime >= 0 && departureTime < 24) { to ++departureTime >= 0.

Increasing the `departureTime` will always make the value to be greater than zero therefore there is no impact on this statement. However since this value is used again in `departureTime<24`, if we have a value that is equal to 23, then the value of 24 will not be less than 24 and the statement returns false instead of true. Since no test suite contains such a test case, this mutant is not killed by any of the test suites. All test suites contain at least one test case with the value of `holiday` equal to true but the value of `departureTime` is not 23 and therefore no test suite is able to kill this mutant.

AOIS_18 changes `departureTime>=0` in

```
if (holiday) {
          if (departureTime >= 0 && departureTime < 24) {
```

to

```
--departureTime >= 0.
```

This is the same case as AOIS_17, except that in order to kill this mutant we need the `departureTime` to be equal to zero for this mutant to get caught. Since the Pair Wise test suite contains such a test case, it is able to kill this mutant. The Base Choice and the Each Choice test suites however are not able to kill this mutant.

AOIS_19 changes `departureTime>=0` to `departureTime++ >= 0` in the same part of the code as AOIS_18. The value of `departureTime` is increased after the first comparison. Therefore if there is a test case with value of `holiday` equal to true and the time is 23, this mutant will be killed. Since no test suite contains such a test case, this mutant is not killed by any of the test suites.

AOIS_20 changes `departureTime>=0` to `departureTime-- >= 0` in the same part of the code as AOIS_18. The value of `departureTime` is decreased after the first comparison. No matter what the value of `departureTime` is it will evaluate to true after the first comparison. Therefore this mutant is equivalent and not killed by any of the test suites.

AOIS_21 changes `departureTime < 24` to `++departureTime < 24` in the same part of the code as AOIS_18.

Since `Sunday` takes priority over a `holiday` we will not reach this part of the code if the day is `Sunday`. If there is a test case where the value of day is a week day and it is a `holiday` and it is 23, this mutant will be killed. All test suites contain a test case where the day is not `Sunday` and is a `holiday` but the time is not 23. Therefore this test case is not killed by any of the test suites.

AOIS_22 changes `departureTime < 24` to `--departureTime < 24` in the same part of the code as AOIS_18.

This test case will be killed if it is a `week day` and a `holiday` and the value of `departureTime` is 24. This will make the program return true when it is in fact false. Since no test suite contains such a test case this mutant is not killed by any of the test suites.

AOIS_23 and AOIS_24 change `departureTime < 24` to `departureTime++ < 24` and `departureTime-- < 24` in the same part of the code as AOIS_18. Since `departureTime` is

never used after this statement, these mutants are equivalent and not killed by any of the test suites.

AOIS_25 changes `departureTime >=0` to `++departureTime >= 0` in

```
if (departureTime >= 0 && departureTime < 7) {
```

This mutant is killed by the Base Choice and the Pair Wise test suites. These test suites contain test cases where increasing the `departureTime` by one has an impact on the other parts of the code. An example is a test case where the `departureTime` is 18 and the day is `Monday`. If the `departureTime` is changed to 19, it will change the `applicableCharge`. This mutant is not killed by the Each Choice test suite because it does not contain such a test case.

AOIS_26 changes the same part of code as AOIS_25 to `--departureTime >= 0`. This mutant is killed by all test suites. These test suites contain a test case where the day is a `weekday` and decreasing the value of `departureTime` will have an impact on the rest of the code leading to a different applicable charge.

AOIS_27 changes the same part of code as AOIS_25 to `departureTime++>= 0`.

This mutant changes the value after comparison. If the value of day is `WeekDay` and the `departureTime` falls into this category the mutant will not be killed. If there is a test case where increasing the `departureTime` by 1 will have an impact on the rest of the code and changes to `applicableCharge`, this mutant will be killed. The Each Choice has a test case where the value of day is `WeekDay` and `departureTime` is not in the specified interval. This test suite however, is not able to kill the mutant because the value of `applicableCharge` will remain the same even if the program enters another if statement. The Base Choice and the Pair Wise test suites contain a test case with such description and are able to kill this mutant.

AOIS_28 changes the same part of code as AOIS_25 to `departureTime-->= 0`. This is the same case as AOIS_27 however this time, the Base Choice test suite does not contain such a test case and is not able to kill this mutant. The Each Choice and the Pair Wise test suites are able to kill this mutant.

AOIS_29 changes the same part of code as AOIS_25 to `++departureTime < 7`. This mutant is killed by the Pair Wise and the Base Choice test suite but not by the Each Choice test suite. The same reasoning as AOIS_27 applies to this mutant.

AOIS_30 changes the same part of code as AOIS_25 to `--departureTime < 7`. This mutant is killed by the Each Choice and the Pair Wise test suites but not by the Base Choice test suite. The same logic as AOIS_28 is applied to this mutant.

AOIS_31 changes the same part of code as AOIS_25 to `departureTime++ < 7`. This mutant is killed by the Base Choice and the Pair Wise test suites but not by the Each Choice test suite because of the same reason as AOIS_27.

AOIS_32 changes the same part of code as AOIS_25 to `departureTime-- < 7`. This mutant is killed by the Each Choice and the Pair Wise test suites but not by the Base Choice test suite. The same logic as AOIS_28 applies to this mutant.

AOIS_33 to AOIS_36 change `departureTime >= 7` in `if (departureTime >= 7 && departureTime < 10) {`

to

```
departureTime => ++departureTime
departureTime => --departureTime
departureTime => departureTime++
departureTime => departureTime-
```

AOIS_37 to AOIS_40 change `departureTime < 10` and `departureTime >= 7` in `if (departureTime >= 7 && departureTime < 10) {`

to:

```
departureTime => ++departureTime
departureTime => --departureTime
departureTime => departureTime++
departureTime => departureTime--
```

AOIS_33, AOIS_35, AOIS_37 and AOIS_39 are killed by the Base Choice and the Pair Wise test suites but not by the Each Choice test suite because of the same logic as AOIS_28. AOIS_34, AOIS_36, AOIS_38 and AOIS_40 are killed by the Each Choice and the Pair Wise test suites but not by the Base Choice test suite because of the same logic as AOIS_29.

AOIS_41 to AOIS_44 change `departureTime >= 10` to `++departureTime, --departureTime, departureTime++, departureTime--` in `if (departureTime >= 10 && departureTime < 17)`.

AOIS_41 and AOIS_43 are killed by the Base Choice and the Pair Wise test suites. This is because these test suites have a test case at the border values and this causes the program to go

into another if statement where the `applicableCharge` is set to a different value. The Each Choice test suite does not contain such a test case.

AOIS_42 is killed by the Base Choice and the Pair Wise test suites because of the same logic as AOIS_41 and AOIS_43.

AOIS_44 is only killed by the Pair Wise test suite. This test suite contains a test case where changing the value of `departureTime` will cause the program to enter another if statement and results in a different applicable charge. The other two test suites do not have such a test case and are not able to kill this mutant. This means having a test frame with a WeekDay and the value of time equal to 17.

AOIS_45 to AOIS_48 change `departureTime < 17` to `++departureTime`, --departureTime, departureTime++ and departureTime-- in if (departureTime >= 10 && departureTime < 17). AOIS_45 is killed by the Base Choice and the Pair Wise test suites. AOIS_46 to aOIS_48 are only killed by the Pair Wise test suites. This is because of the same logic as AOIS_41 to AOIS_43.

AOIS_49 to AOIS_52 change `departureTime >= 17` in `if (departureTime >= 17 && departureTime < 19)` to `++departureTime, -departureTime, departureTime++` and `departureTime--`.
AOIS_49 to AOIS_52 are killed by the Pair Wise test suite but not by the Each Choice and the Base Choice test suites. This is because of the same logic used in `AOIS_40` to `AOIS_49`.

AOIS_53 to AOIS_56 `change departureTime < 19` in `if (departureTime >= 17 && departureTime < 19) {` to `++departureTime, --departureTime, departureTime++` and `departureTime--`.
AOIS_53, AOIS_54 and AOIS_56 are killed by the Pair Wise test suite but not by the Each Choice and the Base Choice test suites. The Pair Wise test suite contains a choice where the value is at the border and changing the value will cause a change in the `applicableCharge`. The Each Choice and the Base Choice test suites do not contain such a test case.

AOIS_55 is not killed by any of the test suites. This is because no test suite contains a test case where the value of day is week day and decreasing the `departureTime` has an impact on the `applicableCharge`.

AOIS_57 to AOIS_60 change `departureTime >= 19` to `++departureTime`, `--departureTime`, `departureTime++` and `departureTime--` in

```
if (departureTime >= 19 && departureTime < 24).
```

AOIS_57 and AOIS_59 are not killed by any of the test suites. No test suite contains a test case where increasing the value of `departureTime` will cause the program to go to another if statement and the `applicableCharge` becomes a different value. AOIS_58 and AOIS_60 are killed by the Pair Wise test suite. The Pair Wise test suite contains a choice with value of day weekday, the `departureTime` is at the border value and changing this value causes a change in the `applicableCharge`. Therefore this test suite is able to kill these mutants. The Base Choice and the Each Choice test suites do not contain such a test case and are not able to kill these mutants.

AOIS_61 to AOIS_64 change `departureTime < 24` to `++departureTime`, `--departureTime`, `departureTime++` and `departureTime--` in

```
if (departureTime >= 19 && departureTime < 24).
```

AOIS_61 is not killed by any of the test suites because no test suite contains a test case where the value of `departureTime` is 23 and the day is a `WeekDay`. AOIS_62 is killed by the Pair Wise test suite because it contains a test case with the value of `departureTime` equal to 24. The mutant will cause the statement to return true in this case instead of false therefore this mutant is killed by this test suite.

AOIS_63 and AOIS_64 are equivalent mutants because the value of `departureTime` is never used after this statement. Therefore they are not killed by any of the test suites.

COI_1 changes `if (destArea != null)` to `if (!(destArea != null))`.
Since there is a test case in all test suites with the value of `destArea` not equal to null, this mutant is killed by all test suites. If the value is not null this will evaluate to false and the value of `destArea` will remain null.

COI_2 changes `if (day == WeekDay.SUNDAY)` to `if (!(day == WeekDay.SUNDAY))`.

Since all test suites contain a test case where the value of day is not `Sunday`, negating the value will cause the test suite to return -1 and null for `applicableCharge`. Therefore this mutant is killed by all test suites.

COI_3 changes `if (departureTime >= 0 && departureTime < 7)` to `if (!(departureTime >= 0) && departureTime < 7).`
This mutant is only killed by the Pair Wise test suite. This is because the Pair Wise test suite is the only one that contains a test case where the value of day is `Sunday` and the `departureTime` is between 0 and 7. The other two test suited do not have such a test case and are therefore not killed by any of the test suites.

COI_4 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime >= 0 && !(departureTime < 7)).`
If a test case has day equal to `Sunday` and the value of `departureTime` greater than 7 or less than 7, this mutant will be killed. The Each Choice and the Pair Wise test suites contain such a test case and therefore kill this mutant. The Base Choice test suite does not contain a test case where the value is `Sunday`. Therefore it is not able to kill this mutant.

COI_5 changes `if (departureTime >= 0 && departureTime < 7)` to `if (!(departureTime >= 0 && departureTime < 7)) {`
This mutant is killed by the Each Choice and the Pair Wise choices. As long as the test suite contains a choice that is `Sunday` and not in this interval, this mutant is killed. Since the Base Choice does not contain a test case with day equal to `Sunday`, it is unable to kill this mutant.

COI_6 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban)).`
The Each Choice and the Base Choice test suites do not cover this part of the code therefore this mutant is not killed by these two test suites. The Pair Wise test suite however reaches this part of the code and therefore is able to kill this mutant.

COI_7 changes if `(destAreaType == suburb)` to `if (!(destAreaType == suburb)).`
Since there are no test cases that reach this part of the code this mutant is not killed by any of the test suites.

COI_8 changes `if (destAreaType == beyondSuburban)` to `if (!(destAreaType == beyondSuburban))`.
Only the Pair Wise test suite reaches this part of the code and therefore is able to kill this mutant.

COI_9 to COI_11 change `if (departureTime >= 7 && departureTime < 24)` to `if (!(departureTime >= 7) && departureTime < 24)`, `if (departureTime >= 7 && !(departureTime < 24))` and `(!(departureTime >= 7 && departureTime < 24))`.
These mutants are killed by the Pair Wise and the Each Choice test suites. If a test case has value of day equal to `Sunday` and the value of `departureTime` is either 7 or not 7 and less than 24, this mutant can be killed. The Pair Wise and the Each Choice test suites are able to kill these mutants since they contain such a test case. The Base Choice test suite is not able to kill them because it does not contain a test case where the value of day is `Sunday`.

COI_12 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban))`.
The Base Choice test suite is not able to kill this mutant because it does not contain a test case where the value of day is `Sunday`.

The Each Choice and the Pair Wise test suites are able to kill this mutant because they both contain a test case where the value of day is `Sunday` and the time is between 7 and 24 and the `destArea` is valid.

COI_13 and COI_14 are not killed by the Each Choice test suite because this test suite does not contain a test case where the day is `Sunday`, the time is between 7 and 24 and the `destArea` is not `urban`.

COI_15 changes `if (holiday)` to `if (!holiday)`.
If a test case contains a value where changing the value of `holiday` will cause a different `applicableCharge`, then this mutant is killed. Since all test suites contain such a test case, this mutant is killed by all test suites.

COI_16 to COI_18 change `if (departureTime >= 0 && departureTime < 24)` to `if (!(departureTime >= 0) && departureTime < 24)`, `if (departureTime >= 0 && !(departureTime < 24))` and `if (departureTime >= 0 && departureTime < 24)`.

All test suites contain a test case where the value of `holiday` is true and the `departureTime` is a correct value. All these mutants cause the program to return false instead of true and vice versa. All mutants are killed by all test suites.

COI_19 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban))`. Since all test suites contain a test case where `holiday` is true, the `departureTime` is correct and the `destAreaType` is correct, this mutant is killed. This is the first line that is checked after entering the loop therefore regardless of the value of `destAreaType`, this mutant is caught.

COI_20 changes `if (destAreaType == suburb)` to `if (!(destAreaType == suburb))`.
Since the Each Choice and the Pair Wise test suites contain a choice where `holiday` is true, the `departureTime` is correct the value of `destinationAreaType` is not `urban`, this mutant is killed. The Base Choice test suite does not contain such a test case and is not able to kill this mutant.

COI_21 changes `if (destAreaType == beyondSuburban)` to `if (!(destAreaType == beyondSuburban))`.
This mutant is killed by the Each Choice and the Pair Wise test suites because they contain a test case where `holiday` is true, the `departureTime` is correct and the `destinationAreaType` is `beyondSuburban`. The Base Choice test suite does not contain such a test case and cannot kill this mutant.

COI_22 to COI_32 make changes to the following statement:

```
if (day == WeekDay.MONDAY || day == WeekDay.TUESDAY || day ==
WeekDay.WEDNESDAY || day == WeekDay.THURSDAY || day == WeekDay.FRIDAY ||
day == WeekDay.SATURDAY) {
```

These mutants insert ! operator in front of each of the statements. The Pair Wise and the Base Choice test suites are able to kill all these mutants. This is because these two test suites contain a test case for every day in the week and it is not a holiday. The Each Choice however is not able to kill COI_22 and COI_31. This is because this test suite does not have a test case where the value of day is equal to Monday or equal to Saturday and is not a holiday. This is a limitation in the number of test cases that the Each Choice test suite contains.

COI_33 to COI_35 make changes to:

```
if (departureTime >= 0 && departureTime < 7).
```

These mutants insert ! operator in front of each of the statements. Since all test suites contain a choice where it is a `weekday` and not a `holiday`, these mutants are killed by all test suites.

COI_36 changes if `(destAreaType == urban)` to `if (!(destAreaType == urban))`. The Pair Wise test suite does not contain a test case where the value of `departureTime` is between 0 and 7, it is a `WeekDay` and not a `holiday`. Therefore this part of the code is never reached. Therefore the mutant is not killed by the Pair Wise test suite. This mutant is killed by the Base Choice and the Each Choice test suites because they do contain such a test case.

COI_37 and COI_38 add `!` to `if (destAreaType == suburb)` and `if (destAreaType == beyondSuburban)`.

These mutants are not killed by any of the test suites because they do not contain a test case where it is a `weekday`, not a `holiday`, between 0 and 7 and is not `urban`.

COI_39 changes `if (departureTime >= 7 && departureTime < 10) {`

to `if (!(departureTime >= 7) && departureTime < 10) {`

This mutant is caught if it is a `weekday` and not a `holiday` and the value is greater than or equal to 7. This mutant is killed by all test suites since they contain such a test case.

COI_40 to COI_41 change the same line as COI_39 to `if (departureTime >= 7 && !(departureTime < 10))` and `if (!(departureTime >= 7 && departureTime < 10))`. These mutants are killed because of the same reason as COI_39.

COI_42 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban))`. Since all test suites contain a test case where it is a `weekday` and not a `holiday` and the value of `departureTime` is greater than or equal to 7, this mutant is killed. The value of `destAreaType` will not make a difference in this case.

COI_43 and COI_44 change `if (destAreaType == suburb)` to `if (!(destAreaType == suburb))`, `if (destAreaType == beyondSuburban)` and `if (!(destAreaType == beyondSuburban))`.

Since the Base Choice test suite does not contain a test case where it is a `weekday` and not a `holiday` and the value of `departureTime` is greater than or equal to 7 and the `destinationAreaType` is not `urban`, these test suites are not killed by this test suite. The other two test suites are able to kill these mutants because they contain such test case.

COI_45 to COI_47 `change if (departureTime >= 10 && departureTime < 17) {` to `if (!(departureTime >= 10) && departureTime < 17) , if (departureTime >= 10 && !(departureTime < 17))` and `if (!(departureTime >= 10 && departureTime < 17))`.

Since all test suites contain a test case where it is a `WeekDay` and not a `holiday` and the `departureTime` is between 10 and 17, these mutants are killed by all test suites.

COI_48 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban))`.
COI_49 changes `if (destAreaType == suburb)` to `if (!(destAreaType == suburb))`.
COI_50 changes `if (destAreaType == beyondSuburban)` to `if (!(destAreaType == beyondSuburban))`.
COI_48 to COI_50 are killed by all test suites because all test suites contain a test case where it is a `weekday` and not a `holiday` and the `departureTime` is between 10 and 17, and the `destAreaType` is either `urban` or `suburb`, or `beyondSuburban`.

COI_51 to COI_53 change `if (departureTime >= 17 && departureTime < 19)` to `if (!(departureTime >= 17) && departureTime < 19), if (departureTime >= 17 && !(departureTime < 19))` and `if (departureTime >= 17 && departureTime < 19) .`

`F4.`

Since all test suites contain a test case where it is a `weekday` and not a `holiday` and the `departureTime` is either between 17 and 19 or is greater than 19, these mutants are killed by all test suites.

COI_54 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban))`. The Each choice test suite does not reach this part of the code and is therefore not able to kill this mutant. The Base Choice and the Each Choice test suites contain a choice reaching this part of the code, and no matter the value of `destAreaType` this mutant will be killed.

COI_55 changes `if (destAreaType == suburb)` to `if (destAreaType == suburb).` Only the Pair Wise test suite contains a test case where it is a `WeekDay` and not a `holiday` and the `departureTime` is between 17 and 19 and the value of `destAreaType` is `suburb`. Therefore this test suite is able to kill this mutant where the Each Choice and the Base Choice are not able to.

COI_56 changes if `(destAreaType == beyondSuburban)` to `if (!(destAreaType == beyondSuburban)).`
Since there are no test cases where the value of `destAreaType` is equal to `suburb` and it is a `WeekDay` and not a `holiday` and the `departureTime` is between 17 and 19, this mutant is not killed by any of the test suites.

COI_57 to COI_59 change `if (departureTime >= 19 && departureTime < 24)` to

```
if (!(departureTime >= 19) && departureTime < 24),
if (departureTime >= 19 && !(departureTime < 24)) and
if (!(departureTime >= 19 && departureTime < 24)).
```

These mutants are killed by all test suites. If a test case has the value of day as `weekday` and the `departureTime` is incorrect or it is in the range of 19 to 24, this mutant will be killed. Since all test suites contain such a test case these mutants are killed by all test suites.

COI_60 changes `if (destAreaType == urban)` to `if (!(destAreaType == urban)).` Since the Each Choice test suite does not contain a test case where the value of day is weekday and the departureTime is between 19 and 24, this part of the code is never reached by this test suite. Therefore this mutant is not killed by the Each Choice test suite.

The Base Choice and the Pair Wise test suites reach this part of the code. No matter the `destAreaType` this mutant will be killed if program reaches this part of the code.

COI_61 changes `if (destAreaType == suburb)` to `if (!(destAreaType == suburb)).`
This mutant is not killed by the Each Choice test suite because of the same reason provided for COI_16. The Base Choice test suite is not able to kill this mutant because it only contains a test case that has `destAreaType` equal to `urban` and never reaches this part of the code. The Pair Wise test suite however is able to kill this mutant because it contains a test case that is in the `departureTime` interval of 19 to 24 and has `destAreaType` equal to `suburb`.

COI_62 changes `if (destAreaType == beyondSuburban)` to `if (!(destAreaType == beyondSuburban))`.

This mutant is not killed by any of the test suites because they do not contain a test case where the time interval is 19 to 24 and the `destAreaType` is equal to `BeyondSuburban`.

COR_1 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime >= 0 || departureTime < 7)`.

Changing `&&` to `||` will cause a different functionality if only one of the two values is true. If there is a test case where the date is `Sunday` and the value of `departureTime` is less than 0 or greater than 7, this statement returns true when it should return false. Since the Each Choice and the Pair Wise test suites contain such a test case, this mutant is killed by them. The Base Choice test suite however is not able to kill this mutant because it does not contain a test case where the day is `Sunday`.

COR_2 changes if (departureTime >= 0 && departureTime < 7) to if (departureTime >= 0 ^ departureTime < 7).

This mutant will be killed if there is a test case with the interval of 0 to 7 or if `departureTime` is less than 0. The Base Choice test suite does not contain a test case with value of day equal to `Sunday` and is therefore not able to kill this mutant. The Each Choice and the Pair Wise test suites however are able to kill this mutant because they contain a test case with the above mentioned description.

COR_3 changes `if (departureTime >= 7 && departureTime < 24)` to `if (departureTime >= 7 || departureTime < 24)`.

If there is a test case where the value of day is `Sunday` and the `departureTime` is greater than 24, this mutant will be killed. This is because a value of greater than 23 will cause this statement to return true since `departureTime` is greater than 7. Since Each Choice test suite contains such test case it is able to kill this mutant. The Pair Wise and the Base Choice however are not able to kill this mutant because they do not contain such test case.

COR_4 changes `if (departureTime >= 7 && departureTime < 24)` to `if (departureTime >= 7 ^ departureTime < 24)`.

If the test suite contains a test case where the value of `departure time` is between 7 and 24 and it is a `Sunday` or the `departureTime` is greater than 24, this mutant will be killed. This is because these inputs will cause the statement to return false instead of true in the first case and true instead of false in the second case. Since the Each Choice and the Pair Wise test suites contain such test case, this mutant is killed. The Base Choice test suite however is unable to kill this mutant because it does not contain a test case with value of day equal to `Sunday`.

COR_5 changes `if (departureTime >= 0 && departureTime < 24)` to `if (departureTime >= 0 || departureTime < 24).`

In order to kill this mutant the value of `holiday` needs to be true and the value of `departureTime` to be either less than zero or greater than 24. Since no test suite contains such a test case, this mutant is not killed by any of the test suites.

COR_6 changes `if (departureTime >= 0 && departureTime < 24)` to `if (departureTime >= 0 ^ departureTime < 24).`

This mutant will be killed if the day is a `holiday` and the `departureTime` is in the interval of 0 to 24. Since all test suites contain such a test case, this mutant is killed by all test suites.

COR_7 changes `if (day == WeekDay.MONDAY || day == WeekDay.TUESDAY || day == WeekDay.WEDNESDAY || day == WeekDay.THURSDAY || day == WeekDay.FRIDAY || day == WeekDay.SATURDAY) {`

to `if (day == WeekDay.MONDAY && day == WeekDay.TUESDAY || day == WeekDay.WEDNESDAY || day == WeekDay.THURSDAY || day == WeekDay.FRIDAY || day == WeekDay.SATURDAY).`

Since all test suites contain a test case with `weekday`, and changing `||` to `&&` causes the program to return false when it is true, this mutant is killed by all test suites.

COR_8 changes `if (day == WeekDay.MONDAY || day == WeekDay.TUESDAY || day == WeekDay.WEDNESDAY || day == WeekDay.THURSDAY || day == WeekDay.FRIDAY || day == WeekDay.SATURDAY)` to

`if (day == WeekDay.MONDAY ^ day == WeekDay.TUESDAY || day == WeekDay.WEDNESDAY || day == WeekDay.THURSDAY || day == WeekDay.FRIDAY || day == WeekDay.SATURDAY).`

This change has no impact on the functionality. This is because no two values can be true at the same time and an `or` is equivalent to `exclusive or`. Therefore this mutant is equivalent and not killed by any of the test suites.

COR_10, COR_12, COR_14 and COR_16 make similar change as COR_8. They are therefore equivalent and not killed by any of the test suites.

COR_9, COR_11, COR_13 and COR_15 make similar changes as COR_7 and are killed by all test suites.

COR_17 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime >= 0 || departureTime < 7).`

If the test suite contains a test case where it is a `WeekDay` and the value of `departureTime` is less than 0 or greater than 7, this mutant will be killed. This is because the expression evaluates to true instead of false. Since all test suites contain such a test case, this mutant is killed by all test suites.

COR_18 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime >= 0 ^ departureTime < 7).`

If a test case is a `weekday` and the `departureTime` is between 0 and 7, this mutant will be killed because the expression will evaluate to false instead of true. A `departureTime` value of less than 0 and greater than 7 will also evaluate to true and kill the mutant. Since all test suites contain such test case, this mutant is killed by all test suites.

COR_19 changes `if (departureTime >= 7 && departureTime < 10)` to `if (departureTime >= 7 || departureTime < 10).`
COR_20 changes `if (departureTime >= 7 && departureTime < 10)` to `if (departureTime >= 7 ^ departureTime < 10).`
COR_21 changes `if (departureTime >= 10 && departureTime < 17)` to `if (departureTime >= 10 || departureTime < 17).`
COR_22 changes `if (departureTime >= 10 && departureTime < 17)` to `if (departureTime >= 10 ^ departureTime < 17).`
COR_23 changes `if (departureTime >= 17 && departureTime < 19)` to `if (departureTime >= 17 || departureTime < 19).`
COR_24 changes `if (departureTime >= 17 && departureTime < 19)` to `if (departureTime >= 17 ^ departureTime < 19).`
COR_25 changes `if (departureTime >= 19 && departureTime < 24)` to `if (departureTime >= 19 || departureTime < 24).`

COR_26 changes `if (departureTime >= 19 && departureTime < 24)` to `if (departureTime >= 19 ^ departureTime < 24).`
COI_19 to COI_26 are killed because of the same logic as COI_17 and COI_18.

LOI_10 and LOI_11 change `if (departureTime >= 0 && departureTime < 7)` to `if (~departureTime >= 0 && departureTime < 7)` and `if (departureTime >= 0 && ~departureTime < 7)`
This mutant will be killed if it is a `WeekDay` and the value of `departureTime` is between 0 and 7 or it is less than 0 or greater than 7. Since all test suites contain such a test case, these mutants are killed.

LOI_12 and LOI_13 change `if (departureTime >= 7 && departureTime < 10)` to `if (~departureTime >= 7 && departureTime < 10)` and `if (departureTime >= 7 && ~departureTime < 10).`

LOI_12 and LOI_13 will be killed if there is a test case that is `WeekDay` and has `departureTime` between 7 and 10. Since all test suites contain such test case these mutants are killed by all test suites.

LOI_14 and LOI_15 change `if (departureTime >= 10 && departureTime < 17)` to `if (~departureTime >= 10 && departureTime < 17)` and `if (departureTime >= 10 && ~departureTime < 17).`

If a test case contains value of `departureTime` in this interval the mutant will be killed. Since all test suites contain such test case, these mutants are killed by all test suites.

LOI_16 and LOI_17 change `if (departureTime >= 17 && departureTime < 19)` to `if (~departureTime >= 17 && departureTime < 19)` and `if (departureTime >= 17 && ~departureTime < 19).`

LOI_16 is killed if there is a test case that reaches this point in the code and the value of `departureTime` is in the interval. LOI_17 will be killed if the value of `departureTime` is greater than 19. If the value is greater than 19 but less than 24 the statement evaluates to true which is incorrect and this mutant will be killed. The Base Choice and the Pair Wise test suites are able to kill LOI_16 because they contain a test case as mentioned above, LOI_17 is killed by all test suites.

LOI_18 and LOI_19 change `if (departureTime >= 19 && departureTime < 24)` to `if (~departureTime >= 19 && departureTime < 24)` and `if (departureTime >= 19 && ~departureTime < 24)`.

The same logic applies to these mutants as LOI_16 and LOI_17. LOI_18 is not killed by the Each Choice test suite but is killed by the Base Choice and Pair Wise test suites. LOI_19 is killed by all test suites.

LOI_4 and LOI_5 change `if (departureTime >= 0 && departureTime < 7)` to `if (~departureTime >= 0 && departureTime < 7)` and `if (departureTime >= 0 && ~departureTime < 7)`.

These mutants are not killed by the Base Choice test suite because it does not contain a test case where the value of day is `Sunday`. The Each Choice test suite is not able to kill LOI_4 because it does not contain a test case with value of `departureTime` in the interval of 0 to 7. LOI_5 is killed by all the Each Choice and Pair Wise test suites. The Pair Wise kills this mutant because it has a test case in the interval of 0 to 7. The Each Choice test suite kills this mutant because it has a value that is not in this interval but the value of the statement will return true causing the value of `applicableCharge` to be incorrect. The Pair Wise test suite has a value that is inside the interval and the second part of the statement will return false when it is true and therefore leads to the wrong `applicableCharge`.

LOI_6 and LOI_7 change `if (departureTime >= 7 && departureTime < 24)` to `if (~departureTime >= 7 && departureTime < 24)` and `if (departureTime >= 7 && ~departureTime < 24)`.

LOI_6 and LOI_7 are not killed by the Base Choice test suite because of the same reason as LOI_4 and LOI_5. LOI_6 is killed by both the Pair Wise and the Each Choice test suites because they contain a test case that is `Sunday` and is in the interval of 7 to 24. The first statement will return false if it is negated and the statement will evaluate to false. This is why this mutant is killed. If the value of `departureTime` is less than 24 then change in LOI_7 will not make a difference in the outcome. This will only be killed if there is a test case with the value of `Sunday` and `departureTime` greater than 23.

LOI_8 to LOI_9 changes `if (departureTime >= 0 && departureTime < 24)` to `if (~departureTime >= 0 && departureTime < 24)` and `if (departureTime >= 0 && ~departureTime < 24)`.

If there is a test case where the value of `holiday` is true and the `departureTime` is between 0 and 24, LOI_8 will be killed. If `holiday` is true and the `departureTime` is greater than 23, LOI_9 will be killed because the second statement evaluates to true instead of false resulting in incorrect value for `applicableCharge`.

ROR_1 changes `if (destArea != null)` to `if (destArea == null)`.
This change will cause the program to go into `destAreaType = destArea.getType()` when the `destArea` is null. Since this is null the `destAreaType` will return null. Therefore this mutant is killed by all test suites since they contain a test case with non-empty values for `destArea`.

ROR_2 changes `if (day == WeekDay.SUNDAY)` to `if (day != WeekDay.SUNDAY)`.
This change will cause the program to go into this statement when the value is a `WeekDay`. If there is a test case where the value is not `Sunday` and the `applicableCharge` will be different than the `applicableCharge` for `Sunday`, then this mutant is killed. Since all test suites contain such test case this mutant is killed by all test suites.

ROR_3 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime > 0 && departureTime < 7)`.
This mutant will be killed only if the day is `Sunday` and the `departureTime` is 0. Since no test case contains these values, this mutant is not killed by any of the test suites.

ROR_4 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime < 0 && departureTime < 7)`.
If the test suite contains a choice that has the day value as `Sunday` and the `departureTime` is between 0 and 7, this mutant will be killed. Also if the test case contains a choice where the value of day is `Sunday` and the `departureTime` is less than zero, this mutant will be killed because the statement will return true which is incorrect. Only the Pair Wise test suite contains a choice where the day is `Sunday` and the `departureTime` is between 0 and 7. Therefore only the Pair Wise test suite is able to kill this mutant.

ROR_5 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime <= 0 && departureTime < 7).`
This mutant is only killed by the Pair Wise test suite because of the same reason provided for ROR_4.

ROR_6 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime == 0 && departureTime < 7).`
This mutant will be killed only if the value of day is `Sunday` and the `departureTime` is not equal to zero and is between 0 and 7. Since only the Pair Wise test suite contains such a test case, only this test suite is able to kill this mutant.

ROR_7 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime != 0 && departureTime < 7).`
This mutant will only be killed if the value of day is `Sunday` and the `departureTime` is equal to zero. Since no test case has a test case with this description, this mutant is not killed by any of the test suites.

ROR_8 changes `if (destAreaType == urban)` to `if (destAreaType != urban).`
There are no test cases in the Base Choice test suite with the day equal to `Sunday`. Therefore this test suite will never reach this part of the code and is unable to kill this mutant. This mutant will be killed if the day is `Sunday` and the `departureTime` is between 0 and 7. The Pair Wise test suite contains such test case and is able to kill this mutant. The Each Choice test suite does not contain such test case therefore it never reaches this part of the code and is unable to kill this mutant.

ROR_9 changes `if (destAreaType == suburb)` to `if (destAreaType != suburb).`
If there is a test case where the day is `Sunday` and the `departureTime` is between 0 and 7 and `destAreaType` is not `urban`, this mutant will be killed. There are no such test cases in any of the test suites therefore this mutant is not killed.

ROR_10 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban).`

This mutant will only be killed if there is a test case where the day is `Sunday` and the `departureTime` is between 0 and 7 and `destAreaType` is `beyondSuburban`. There are no such test cases in any of the test suites therefore this mutant is not killed.

ROR_11 to ROR_15 change `if (departureTime >= 7 && departureTime < 24)` to `(departureTime > 7 && departureTime < 24)`, `(departureTime < 7 && departureTime < 24)`, `if (departureTime != 7 && departureTime < 24)` and `if (departureTime == 7 && departureTime < 24)`.
If there is a test case where the value of day is `Sunday` and the `departureTime` is 7, ROR_11 to ROR_14 will be killed. The Pair Wise and the Each Choice test suites contain such test case and are therefore able to kill these mutants. The Base Choice test suite does not contain any test case that is `Sunday` and therefore never gets to this part of the code and is unable to kill these mutants.

ROR_15 can only be killed if the value of `departureTime` is not 7 and is less than 24. Since only the Pair Wise test suite contains such a test case, it is able to kill ROR_15. Other two test suites are unable to kill this mutant.

ROR_16 changes `if (destAreaType == urban)` to `if (destAreaType != urban)`.
This mutant will be killed if there is a test case that has weekday as `Sunday`, has a `departureTime` between 7 and 24. Since the Each Choice and the Pair Wise test suites contain such a test case, this mutant is killed.

ROR_17 changes if `(destAreaType == suburb)` to `if (destAreaType != suburb)`.
If there is a t test case that has weekday as `Sunday`, has a `departureTime` between 7 and 24 and is not `urban`, this mutant will be killed. Since only the Pair Wise test suite has such a test case, this test suite is able to kill this mutant while the other two test suites are not able to kill it.

ROR_18 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban)`.
If there is a t test case that has weekday as `Sunday`, has a `departureTime` between 7 and 24 and is `beyondSuburban`, this mutant will be killed. Since only the Pair Wise test suite has such a

test case, this test suite is able to kill this mutant while the other two test suites are not able to kill it.

ROR_19 changes `if (departureTime >= 0 && departureTime < 24)` to `if (departureTime > 0 && departureTime < 24).`
This mutant will only be killed if it is a `holiday` and the `departureTime` is equal to zero. Since only the Pair Wise test suite has such a test case, this test suite is able to kill this mutant while the other two test suites are not able to kill it.

ROR_20 to ROR_23 change `if (departureTime >= 0 && departureTime < 24)` to

```
if (departureTime < 0 && departureTime < 24),
if (departureTime <= 0 && departureTime < 24),
if (departureTime == 0 && departureTime < 24) and
if (departureTime != 0 && departureTime < 24).
```

If it is a `holiday` and the `departure time` is between 0 and 24, ROR_20 and ROR_21 will be killed. This is because the first part of the program evaluates to false even though the value of `departureTime` lies in the interval. All test suites contain such a test case and are therefore able to kill these two mutants.

If a test case contains a choice that is not zero and is less than `departureTime` and is a `holiday`, ROR_22 will be killed. All test suites contain such a test case and are therefore able to kill this mutant.

ROR_23 will be killed if it is a `holiday` and the value of `departureTime` is zero or less than zero. Since only the Pair Wise test suite contains such a test case, this test suite is able to kill this mutant. The Base Choice and the Each Choice test suites are not able to kill this mutant because they do not have such a test case.

ROR_24 changes `if (destAreaType == urban)` to `if (destAreaType != urban).`
If a test case is a `holiday` and the value of `departureTime` is between 0 and 24 and the `destAreaType` is `urban`, this mutant will be killed. This is because the value of `applicableCharge` will be different than the actual value. Since all test suites contain such a test case this mutant is killed by all test suites.

ROR_25 changes `if (destAreaType == suburb)` to `if (destAreaType != suburb).`

If a test case is a `holiday` and the value of `departureTime` is between 0 and 24 and the `destAreaType` is not `urban`, this mutant will be killed. The Pair Wise and the Each Choice test suites contain such a test case and are able to kill this mutant. The Base Choice test suite does not contain such test case and is unable to kill this mutant.

`ROR_26` changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban).`

If a test case is a `holiday` and the value of `departureTime` is between 0 and 24 and the `destAreaType` is `beyondSuburban`, this mutant will be killed. Since the Pair Wise and the Each Choice test suites contain such test case they are able to kill this mutant. The Base Choice however is not able to kill this mutant because it does not contain such a test case.

ROR_27 to ROR_32 change the `==` in `if (day == WeekDay.MONDAY || day == WeekDay.TUESDAY || day == WeekDay.WEDNESDAY || day == WeekDay.THURSDAY || day == WeekDay.FRIDAY || day == WeekDay.SATURDAY)` to `!=`.

These mutants will be killed if there is a test case that is not a `holiday` and equal to the date that is being changed. The Each Choice cannot kill ROR_27, ROR_29 and ROR_32 because it does not contain a test case where it is not a holiday and is `Monday`, `Friday` or `Saturday`. The Base Choice test suite is able to kill all mutants because it contains a test case for all days and is not a `holiday`. The Pair Wise Test suite is able to kill all these mutants because it contains test cases for each of the days and is not a `holiday`.

ROR_33 changes `if (departureTime >= 0 && departureTime < 7)` to `if (departureTime > 0 && departureTime < 7).`

If a test case is not a `holiday` and is a `WeekDay` and the value of `departureTime` is equal to zero this mutant will be killed. Since the Base Choice test suite contains such test case it is able to kill this mutant. The Each Choice and the Pair Wise test suites are not able to kill this mutant because they do not contain such test case.

ROR_34 to ROR_37 change `if (departureTime >= 0 && departureTime < 7)` to

```
if (departureTime < 0 && departureTime < 7),
if (departureTime <= 0 && departureTime < 7),
if (departureTime == 0 && departureTime < 7) and
if (departureTime != 0 && departureTime < 7).
```

ROR_34 and ROR_35 will be killed if a test case is not a `holiday` and is a `weekday` and the value of `departureTime` is between 0 and 7. This is because the change to `<` and `<=` will cause the statement to return false if `departureTime` is in the interval. If `departureTime` is also less than zero the mutants will be caught. Since all test suites contain such a test case, these mutants are killed by all test suites.

ROR_36 requires a test case where it is not a `holiday`, is a `WeekDay` and the value of `departureTime` is not equal to zero. Only the Each Choice test suite contains such a test case and is able to kill this mutant. The Pair Wise and the Base Choice test suites are unable to kill this mutant because they do not have such a test case.

ROR_37 will be killed if it is not a `holiday`, is a `weekday` and the value of `departureTime` is equal to zero or less than 0 and less than 7. Since all test suites contain such a test case, this mutant is killed by all test suites.

ROR_38 changes `if (destAreaType != urban)` to `if (destAreaType == urban).`
If a test case reaches this part of the code, meaning it is not a `holiday`, is a `weekday` and the value of `departureTime` is between 0 and 7. This mutant will be killed. Since the Base Choice and the Each Choice test suites contain such a test case they are able to kill this mutant. The Pair Wise test suite however does not have a test case with the description and is not able to kill this mutant.

ROR_39 changes `if (destAreaType == suburb)` to `if (destAreaType != suburb).`
If a test case reaches this part of the code, meaning it is not a `holiday`, is a `weekday` and the value of `departureTime` is between 0 and 7 and the `destAreaType` is not urban, this mutant will be killed. No test suite contains such a test case and therefore this mutant remains alive.

ROR_40 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban).`
If a test case reaches this part of the code, meaning it is not a `holiday`, is a `weekday and the value of departureTime` is between 0 and 7 and the `destAreaType` is not `urban` or `suburb`, this mutant will be killed. Since no test suite contains such test case, this mutant is not killed.

ROR_41 changes `if (departureTime >= 7 && departureTime < 10)` to `if (departureTime > 7 && departureTime < 10)` .

If there is a test case where it is not a `holiday` and the value of `departureTime` is 7, this mutant will be killed. No test suite contains such test case and therefore this mutant is not killed.

ROR_42 to ROR_45 changes `if (departureTime >= 7 && departureTime < 10)` to

```
if (departureTime < 7 && departureTime < 10),
if (departureTime <= 7 && departureTime < 10),
if (departureTime == 7 && departureTime < 10) and
if (departureTime != 7 && departureTime < 10).
```

If there is a test case where it is not a holiday and the value of departureTime is between 7 and 10, or the departureTime is less than zero, ROR_42 and ROR_43 will be killed. If this test case has a departureTime that is not equal to 7 and less than 10, ROR_44 will be killed and if the departureTime is less than 0 or equal to 7, ROR_45 will be killed. Since all test suites contain a test case with the above mentioned description, these mutants are killed by all test suites.

ROR_46 changes `if (destAreaType == urban)` to `if (destAreaType != urban).` This mutant is killed if there is a test case where it is not a `holiday` and the value of `departureTime` is between 7 and 10 and the `destAreaType` is equal to `urban`. Since all test suites contain such a test case, this mutant is killed by all test suites.

ROR_47 changes `if (destAreaType == suburb)` to `if (destAreaType != suburb).` This mutant is killed if there is a test case where it is not a `holiday` and the value of `departureTime` is between 7 and 10 and the `destAreaType` is not equal to `urban`. Since only the Each Choice and the Pair Wise test suites contain such test case, this mutant is killed by these two test suites. The Base Choice test suite is unable to kill this mutant because it does not contain such test case.

ROR_48 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban).`

This mutant is killed if there is a test case where it is not a `holiday` and the value of `departureTime` is between 7 and 10 and the `destAreaType` is not equal to `urban` or `suburban`. This mutant is killed by the Each Choice and the Pair Wise test suites because they contain such test case. The Base Choice test suite is unable to kill this mutant.

ROR_49 to ROR_53 change `if (departureTime >= 10 && departureTime < 17)` to
`if (departureTime > 10 && departureTime < 17),`
`if (departureTime < 10 && departureTime < 17),`
`if (departureTime <= 10 && departureTime < 17),`
`if (departureTime == 10 && departureTime < 17)` and
`if (departureTime != 10 && departureTime < 17).`
The first condition for these mutants to be killed is if the test case is a `holiday` and a `WeekDay`.

ROR_49 is killed if the value of `departureTime` is equal to 10. ROR_50 and ROR_51 are killed if the value of `departureTime` is less than 0. ROR_52 is killed if there is a test case that the value of `departureTime` is not equal to 10 and less than 17 or is less than 0. ROR_53 is killed if the value of `departureTime` is equal to 10.

ROR_49 is only killed by the Each Choice test suite; ROR_50, ROR_51 and ROR_53 are killed by all test suites. ROR_52 is killed by the Base Choice and the Pair Wise test suites. This is because these test suites contain a test case described above.

ROR_54 changes `if (destAreaType == urban)` to `if (destAreaType != urban).`

This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 10 and 17. The value of `destAreaType` does not make a difference. All test suites contain such a test case and are able to kill this mutant.

ROR_55 changes `if (destAreaType == suburb)` to `if (destAreaType != suburb).`

This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 10 and 17 and the `destAreaType` is not `urban`. All test suites contain such a test case and are able to kill this mutant.

ROR_56 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban).`

This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 10 and 17 and the `destAreaType` is not `urban` or `suburban`. All test suites contain such test case and can kill this mutant.

ROR_57 to ROR_61 change `if (departureTime >= 17 && departureTime < 19)` to
`if (departureTime > 17 && departureTime < 19),`
`if (departureTime < 17 && departureTime < 19),`

```
if (departureTime <= 17 && departureTime < 19),
if (departureTime >= 17 && departureTime < 19) and
if (departureTime >= 17 && departureTime < 19).
```

The first condition for these mutants to be killed is if the test case is a `holiday` and a `weekday`.

ROR_57 is killed if the value of `departureTime` is equal to 17. ROR_50 and ROR_51 are killed if the value of `departureTime` is less than 0. ROR_52 is killed if there is a test case that the value of `departureTime` is not equal to 17 and less than 19 or is less than 0. ROR_53 is killed if the value of `departureTime` is equal to 17.

ROR_57 is only killed by the Pair Wise test suite, ROR_58, ROR_59 and ROR_61 are killed by all test suites. ROR_60 is killed by the Base Choice and the Pair Wise test suites. This is because these test suites contain a test case described above.

ROR_62 changes `if (destAreaType == urban)` to `if (destAreaType != urban).`
This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 17 and 19. The value of `destAreaType` does not make a difference. This mutant is killed by the Base Choice and the Pair Wise test suites because they contain such test case.

ROR_63 changes `if (destAreaType == suburb) to if (destAreaType != suburb).`
This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 17 and 19 and the `destAreaType is not urban`. Only the Pair Wise test suite kills this mutant because it contains such test case.

ROR_64 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban).`
This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 17 and 19 and the `destAreaType` is not `urban` or `suburban`. This mutant is not killed by any test suite because not test suite contains such test case.

ROR_65 to ROR_69 change `if (departureTime >= 19 && departureTime < 24)` to

```
if (departureTime > 19 && departureTime < 24),
if (departureTime < 19 && departureTime < 24),
if (departureTime <= 19 && departureTime < 24),
if (departureTime == 19 && departureTime < 24) and
if (departureTime != 19 && departureTime < 24).
```

The first condition for these mutants to be killed is if the test case is a `holiday` and a `weekday`.

ROR_65 is killed if the value of departureTime `is` equal to 19. ROR_66 and ROR_67 are killed if the value of `departureTime` is less than 0. ROR_68 is killed if there is a test case that the value of departureTime is not equal to 19 and less than 24 or is less than 0. ROR_69 is killed if the value of `departureTime` is equal to 19.

ROR_65 is only killed by the Pair Wise test suite. ROR_66, ROR_67 and ROR_69 are killed by all test suites. ROR_68 is killed by the Base Choice. This is because these test suites contain a test case described above.

ROR_70 changes `if (destAreaType == urban)` to `if (destAreaType != urban)`.

This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 19 and 24. The value of `destAreaType` does not make a difference. This mutant is killed by the Base Choice and the Pair Wise test suites because they contain such test case.

ROR_71 changes `if (destAreaType == suburb)` to `if (destAreaType != suburb)`.

This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 19 and 24 and the `destAreaType` is not `urban`. Only the Pair Wise test suite kills this mutant because it contains such test case.

ROR_72 changes `if (destAreaType == beyondSuburban)` to `if (destAreaType != beyondSuburban)`.

This mutant is killed if the test case is a `holiday` and a `weekday` and the value of `departureTime` is between 19 and 24 and the `destAreaType` is not `urban` or `suburban`. This mutant is not killed by any test suite because not test suite contains such test case.

The function `populateInMemory` is never used for testing purposes therefore the mutants generated are not killed by any of the test suites.