Swansea University

Computer Science Department
CS 339
Supervisors: Dr. M. Roggenbach, Prof. Holger Schlingloff

Equivalence Class Testing

Garreth Davies

12/01/07

Abstract

This document describes techniques used by testers to test computer software. It primarily focuses on the concept of Equivalence Class testing, but also gives an insight into boundary-value testing and decision-based table testing. A brief overview of functional testing is also mentioned.

1 Introduction

One of if not the most important steps in producing good-rate software is testing. Testing is often overlooked as an almost unnecessary step in the software process. However software developers are now realizing the vital role that testing plays in securing successful software. More time and money is being spent on testing software. It is cheaper for a company to test their software thoroughly than to spend less time on testing and find out they must recall their software and re-design it because of faulty code. The U.S. Commerce Department has estimated that buggy software costs nearly $60 billion annually, and that at least $22.2 billion worth of those bugs could have been prevented with more thorough testing [1].

Testing however can be split up into two key areas

1. Functional Testing (Black Box) and
2. Structural Testing (White Box).

Structural testing focuses on what a system is composed of and how it is structured. However this paper will focus on aspects of functional testing and will not delve any further into structural testing.

The principal aim behind functional testing is to assess how well a system executes the functions it is suppose to. This could be anything from how well a GUI application works to how a particular search performs. Functional Testing takes an external perspective of the system and carries out tests based on input into a function and output out of that function. Comparing an expected output with the output produced by the function will determine whether the test was successful.

Within functional testing are three different kinds of testing

- Boundary-Value testing,
- Equivalence Class testing and
- Decision Table-Based testing.

The focus of this paper will be on Equivalence Class testing, which is a step beyond Boundary-value testing and a step behind decision based table testing in terms of sophistication. Equivalence Class testing is a powerful method of testing since it cuts down the number of test cases required to test a system reasonably with regards to Boundary-Value testing.

2 Background

2.1 What is Equivalence Class Testing?

Equivalence class testing is the next logical step in our model of functional testing. This technique improves the quality of test cases by removing the vast amounts of redundancy and gaps that appeared in boundary-value testing. The fundamental principal behind this method is the formation of Equivalence classes.

Input or output data is grouped or partitioned into sets of data that we expect to behave similarly using an Equivalence relation. An equivalence relation describes how data is going to be processed when it enters a function. It is here where most of the time and effort is required in equivalence class testing. Creating a strong equivalence relation will result in optimal and useful test cases, on the contrary a poorly designed equivalence relation will often result in surplus test cases with inadequate results.

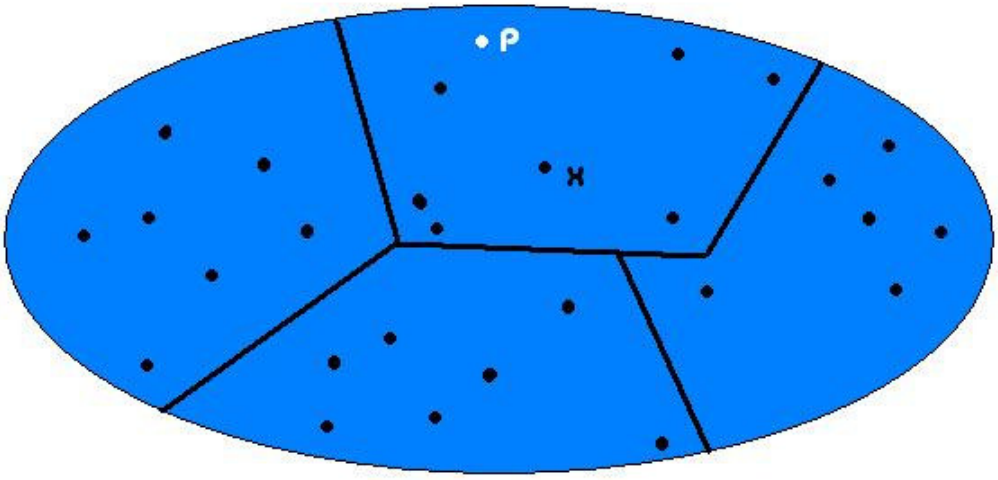

Figure 2.1

This diagram (Figure 2.1) represents our set of data (input or output) that we wish to use to produce test cases. The data is split up into subsets, each subset containing some of the original data. The two points to note here would be the facts that the whole set is being included ensuring notion of completeness and the disjoint ness between subsets ensures non redundancy. The way in which these subsets are created depends on the type of equivalence relation used. Now that we have split our data up into groups we can carry out our test cases.

Since we have used an equivalence relation to split the data we can safely make the assumption that all data within a subset will behave similarly. Thus instead of testing the whole subset we can just take one point, **p** and say that:

If we produce a test case using point p then the result will be the same as all the points x1 to xn within the subset.

From this property of equivalence class testing we can see that redundancy is greatly reduced, since only one point from each subset needs to be "checked". It is this property that produces Equivalence Class Testings biggest advantage. We are able to greatly reduce redundancy and improve our test cases by eliminating gaps.

3 Applications of Equivalence Class Testing

There are two main properties that underpin the methods used in functional testing. The single fault assumption and the multiple fault assumption. These two properties lead to two different types of equivalence class testing, weak and strong. However if we decide to test for invalid input/output as well as valid input/output we can produce another two different types of Equivalence Class Testing, normal and robust. Robust Equivalence Class testing takes into consideration the testing of invalid values, whereas normal does not. Therefore we now have four different types of Equivalence Class Testing, namely weak normal, strong normal, weak robust and strong robust.

3.1 Weak Normal Equivalence Class Testing

Weak equivalence class testing is based on the single fault assumption, stating that rarely is an error caused as a result of two or more faults occurring simultaneously. Therefore weak equivalence class testing only takes one variable from each equivalence class (figure 3.1).

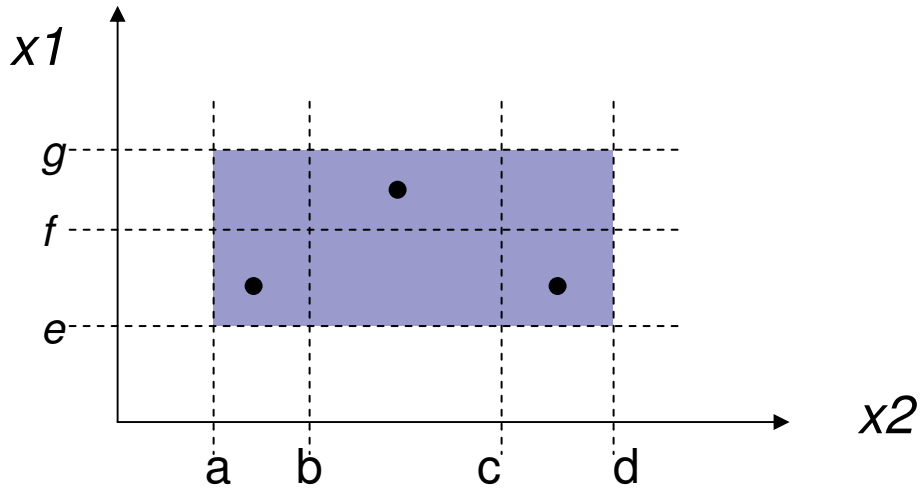

Figure 3.1

3.2 Strong Normal Equivalence Class Testing

Conversely Strong Equivalence Class testing is based on the multiple assumption which states that errors will result in a combination of faults. Therefore strong equivalence class testing tests every combination of elements formed as a result of the Cartesian product of the Equivalence relation (figure 3.2).

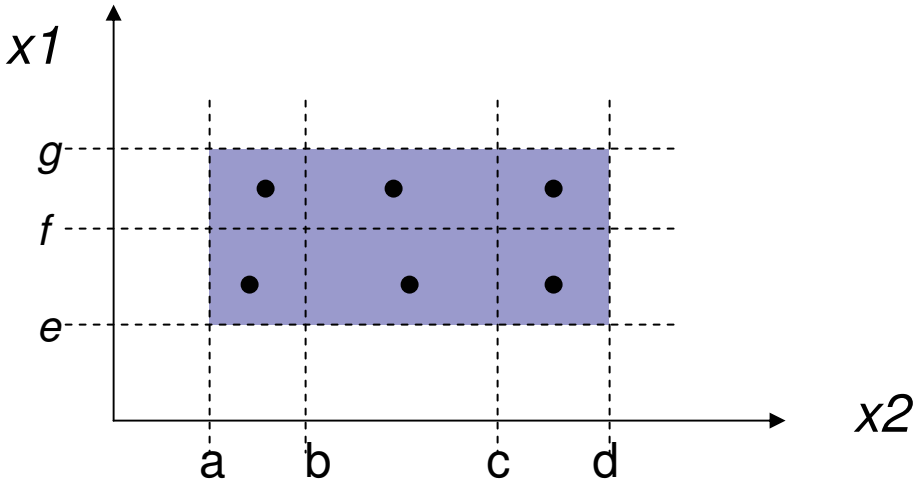

Figure 3.2

3.3 Weak Robust Equivalence Class Testing

As with weak normal Equivalence Class testing we only test for one variable from each Equivalence Class. However we now also test for invalid values as well. Since weak Equivalence Class Testing is based on the single fault assumption a test case will have one invalid value and the remaining values will all be valid.

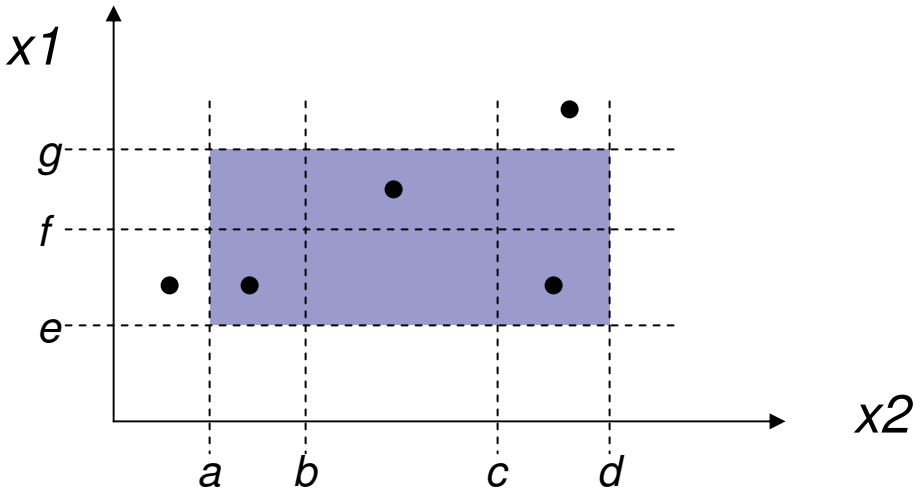

Figure 3.3

3.4 Strong Robust Equivalence Class Testing

This form of Equivalence Class testing produces test cases for all valid and invalid elements of the Cartesian product of all the equivalence classes.

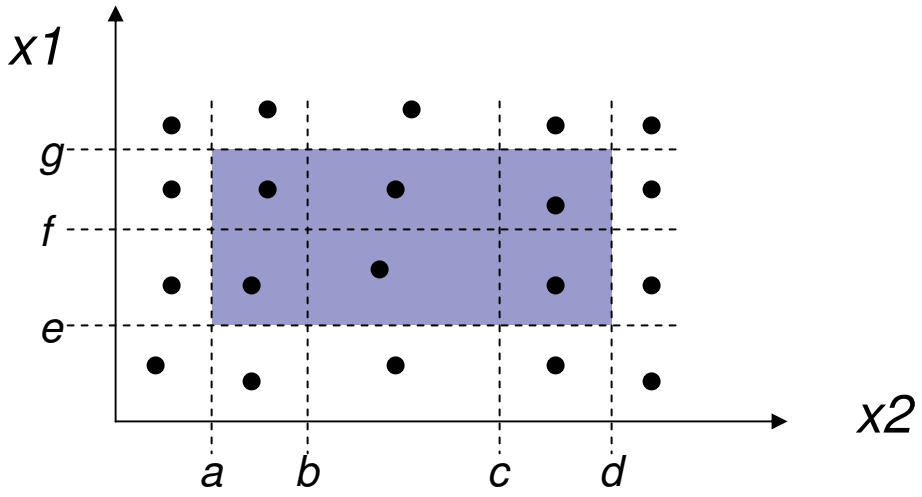

Figure 3.4

As seen from the diagrams above there are problems when moving from weak Equivalence Class testing to strong. The main problem being that we encounter massive redundancy, especially during strong robust equivalence class testing. This takes us a step backwards towards boundary-value testing, the whole concept behind equivalence class testing is to produce useful test cases while reducing redundancy. During strong robust equivalence class testing the first is usually achieved while the latter is not.

There are other problems with robust equivalence class testing that hinder the overall progress of testing. Testers use a specification which tells them what the expected outputs should be when entering test cases into a system. More often than not the specification will not identify expected outputs for invalid test cases. This means that testers have to spend excessive time defining expected outputs for invalid test cases, slowing the overall progress of the testing process down.

Another vital problem of robust equivalence class testing is that it might in effect be a waste of time. The birth of strongly typed languages has discarded the need to test for invalid inputs.

4 The Equivalence Relation

The equivalence relation as stated earlier is a vital aspect to equivalence class testing. The whole process of producing optimal and useful test cases is pivoted around creating a meaningful equivalence relation. To understand this view we will look at the NextDate Function as illustrated in the book, A Craftsman's Approach [2].

4.1 An Example - The NextDate Function

The NextDate function is a function which will take in a date as input and produces as output the next date in the Georgian calendar. It uses three variables (day, month and year) which each have valid and invalid intervals.

4.1.1 First Attempt

A first attempt at creating an equivalence relation might produce intervals such as these:

Valid Intervals

M1 = { month : 1 ≤ month ≤ 12 }
D1 = { day : 1 ≤ day ≤ 31 }
Y1 = { year : 1812 ≤ year ≤ 2012 }

Invalid Intervals

M2 = { month : month < 1 }
M3 = { month : month > 12 }
D2 = { day : day < 1 }
D3 = { day : day > 31 }
Y2 = { year : year < 1812 }
Y3 = { year : year > 2012 }

At a first glance it seems that everything has been taken into account and our day, month and year intervals have been defined well. Using these intervals we produce test cases using the four different types of Equivalence Class testing.

Weak and Strong Normal

Day	Month	Year	Expected Output
15	6	1912	16/6/1912

Figure 4.1

Since the number of variables is equal to the number of valid classes, only one weak normal equivalence class test case occurs, which is the same as the strong normal equivalence class test case (figure 4.1).

Weak Robust

Day	Month	Year	Expected Output
15	6	1912	16/6/1912
-1	6	1912	day not in range
32	6	1912	day not in range
15	-1	1912	month not in range
15	13	1912	month not in range
15	6	1811	year not in range
15	6	2013	year not in range

Figure 4.2

Here (figure 4.2) we can see that weak robust equivalence class testing will just test the ranges of the input domain once on each class. Since we are testing weak and not normal, there will only be at most one fault per test case (single fault assumption) unlike Strong Robust Equivalence class testing.

Strong Robust

This is a table showing one corner of the cube in 3d-space (the three other corners would include a different combination of variables) since the complete table would be too large to show.

Day	Month	Year	Expected Output
15	-1	1912	Value of month not in the range 1..12
-1	6	1912	Value of day not in the range 1..31
15	6	1811	Value of year not in the range 1812..2012
-1	-1	1912	Value of month not in the range 1..12
			Value of day not in the range 1..31
-1	6	1811	Value of day not in the range 1..31
			Value of year not in the range 1812..2012
15	-1	1811	Value of month not in the range 1..12
			Value of year not in the range 1812..2012
-1	-1	1811	Value of month not in the range 1..12
			Value of day not in the range 1..31
			Value of year not in the range 1812..2012

Figure 4.3

The multiple fault assumption is clearly visible in these test cases produced by strong robust equivalence class testing (figure 4.3). Not only can one fault be responsible for producing an invalid input, it can also be a combination of two and finally all three variables.

The previous test cases produced by our original equivalence relation are inadequate. They do not address the NextDate function problem satisfactorily. We get information from the test cases such to the limits of a year, month and day, but there is no information specific to a Georgian style calendar. This is due to poor design when we created our equivalence relation. In our new design we must think about questions such as:

- What happens when a month only has 30 days?
- What happens if the month is February?
- What happens if we have a leap year and the month is February?

4.1.2 Second Attempt

As said before the equivalence relation is vital in producing useful test cases and more time must be spent on designing it. If we focus more on the equivalence relation and consider more greatly what must happen to an input date we might produce the following equivalence classes:

M1 = { month : month has 30 days }
M2 = { month : month has 31 days }
M3 = { month : month is February }

Here month has been split up into 30 days (April, June, September and November), 31 days (January, March, April, May, July, August, October and December) and February.

D1 = { day : $1 \leq$ day ≤ 28 }
D2 = { day : day = 29 }
D3 = { day : day = 30 }
D4 = { day : day = 31 }

Day has been split up into intervals to allow months to have a different number of days, we also have the special case of a leap year (February 29 days).

Y1 = { year : year = 2000 }
Y2 = { year : year is a leap year }
Y3 = { year : year is a common year }

Year has been split up into common years, leap years and the special case the year 2000 so we can determine the date in the month of February.

Here are the test cases for the new equivalence relation using the four types of Equivalence Class testing.

Weak Normal

Day	Month	Year	Expected Output
14	6	2000	15/6/2000
29	7	1996	30/7/1996
30	2	2002	impossible date
31	6	2000	impossible input date

Figure 4.4

Strong Normal

Day	Month	Year	Expected Output
14	6	2000	15/6/2000
14	6	1996	15/6/1996
14	6	2002	14/6/2002
29	6	2000	30/6/2000
29	6	1996	30/6/1996
29	6	2002	30/6/2002
30	6	2000	1/6/1996
30	6	1996	1/6/1992
...	...	...	...
30	2	2002	impossible date
31	2	2000	impossible date
31	2	1996	impossible date
31	6	2002	impossible date

Figure 4.5

Looking at our test cases here we can see that we have impossible dates as output. This is because Equivalence class testing uses automatic test generation, meaning values are selected automatically, roughly from the middle of each class. Due to values being automatically selected we can see how this process has no knowledge of a Georgian calendar. Dates that would be interesting and useful to a tester would be on leap years in February where the date is the 28th or 29th. Other useful dates would be at the end of months to test whether the months have the correct number of days in them. Automatic test generation does not take these kinds of details into consideration and so this will always be a problem.

As noted previously the move from weak to normal testing produces vast amounts of redundancy. If we were to show the Strong robust equivalence class test cases then we would have 150 test cases!

Although this equivalence relation is not perfect it is a significant improvement on our first attempt, we can now relate our Nextdate function to the Georgian calendar. This simple example illustrates how crucial it can be to produce a high quality Equivalence Relation.

5 Functional Testing Overview

As mentioned earlier within functional testing there are three different types of testing, each with their own advantages and disadvantages. These can be illustrated by the diagrams below.

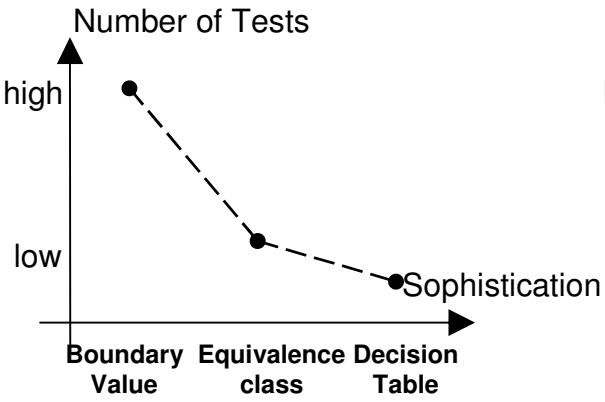

Figure 5.1

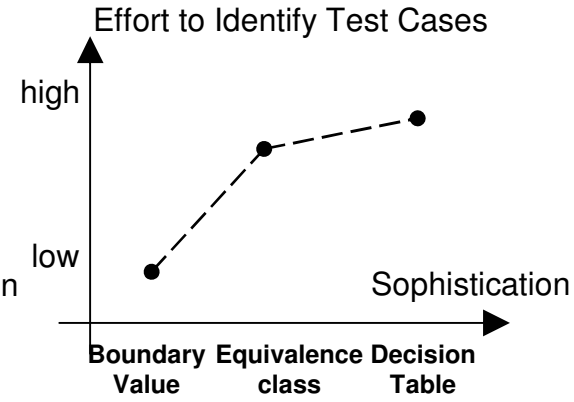

Figure 5.2

Figure 5.1 shows the relationship between the numbers of test cases produced by a testing type against the level of sophistication of that testing type. Sophistication describes how useful the test cases produced by the testing type are in testing a particular function. A high level of sophistication in a testing type would result in test cases which prove very useful to the tester.

Equivalence Class testing produces a relatively low number of test cases while maintaining a sufficient level of sophistication. Equivalence class testing improves greatly on Boundary-Value testing by reducing the number of test cases. Although Equivalence class testing is not as sophisticated as Decision-Based table it does have some advantages over it, as illustrate in the figure 5.2.

Figure 5.2 shows the relationship between the effort needed to identify test cases and the level of sophistication for a particular testing type. Boundary Value testing requires the least effort to identify and produce test cases, which is advantageous if the tester wishes to spend little time on testing. However as noted earlier the test cases produced will be inadequate and large in number.

Equivalence Class testing requires a relatively large amount of effort to identify test cases. This is due to the time and effort that must be spent on creating a good equivalence relation. So while Equivalence Class testing is not as

sophisticated as Decision-based table testing, it does require less effort to implement.

6 Conclusion

Testing proves to be a vital role in the development of any software and is becoming increasingly popular with software developers. Mistakes made by companies teach us that if software is going to be successful it must be extensively tested before its release.

A large portion of testing falls under functional testing which focuses on the input and output of a system and how they are related. Usually named black box testing, functional testing does not put its focus on the implementation of a system but rather on the actions it executes.

Equivalence class testing along with boundary-value testing and decision-table based testing make up the basis of functional testing. Equivalence class testing partitions the input/output of a system. Contained within these subsets will be elements, within each subset all elements will be "treated the same" by the system. Using this assumption Equivalence Class testing need only test one element from each subset to prove that all elements within the set are correct. It is this property of Equivalence class testing that makes it so appealing. However in order to implement Equivalence class testing to its full potential some time must be spent on designing a strong equivalence relation.

6.1 Further Work

The knowledge gained during the length of the course will become extremely valuable when it comes to developing future software. I plan use this knowledge to test my prototype software that I am developing for my CS-344 course. The software that I am developing for this course will benefit from the kind of functional testing that I have been studying. I believe Equivalence Class testing will prove a very useful tool in terms of strengthening my software.

Parts of my software will require input from a user to produce results in the form of tables and charts. Equivalence class testing could be used to test this kind of input. Some time would have to be spent on determining a good equivalence relation but this would be time well spent as the program would benefit greatly.

7 References

[1] – Software QA 101: The Basics of Testing > Functional Testing
http://www.awprofessional.com/articles/article.asp?p=333473&rl=1
Hildreth, Sue – 03/09/04
Last accessed on 30/12/06

[2] – Software Testing: A Craftsman's Approach, 2nd Edition
Jogensen, Paul C – CRC Press July 2002

[3] – The Art of Software Testing, 2nd Edition
John Wiley & Sons Canada, Ltd – June 2004