An Evaluation of Systematic Functional Testing Using Mutation Testing

Steve Linkman
Computer Science Department
Keele University – United Kingdom
s.g.linkman@cs.keele.ac.uk

Auri Marcelo Rizzo Vincenzi[*]
José Carlos Maldonado
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo – Brazil
{auri, jcmaldon}@icmc.usp.br

Abstract

We describe a criterion named *Systematic Functional Testing* that provides a set of guidelines to help the generation of test sets. The test sets generated by the Systematic Functional Testing and by other functional approaches and Random Testing are compared against Mutation Testing. The effectiveness of each test set is measured based on the mutation score using *PROTEUM/IM* 2.0– a mutation testing tool. We conducted a case study using the *Cal* UNIX programme. The test set generated by the Systematic Functional Testing criterion killed all the non-equivalent mutants while the others approaches scored significantly less.

Keywords: Functional Testing, Systematic Functional Testing, Mutation Testing, Software Testing.

1 Introduction

When we try to test software we have to ensure that the behaviour of the programme matches the planned behaviour. In the literature we find proposals to do this in a number of ways, these include, structured testing, functional testing, random testing, data driven testing and others. Traditional wisdom indicates that we should undertake structural testing aiming to get the various coverage measures of the programme as high as possible. However such an approach is expensive and is at best a substitute measure as it does not look at behaviour. We would propose that the use of mutation testing as a measure of the effectiveness of a test set in finding errors in a programme. In this case a test suite that killed 100% of all mutants is going to ensure the correct behaviour of the programme, given the mutant generation is effective.

On this premise we set out to test the effectiveness of various approaches to test generation by using them on the same programme, in this case the programme *Cal* in UNIX which is used to generate calendars based on the input parameters. The approaches we assessed were:

- Functional testing as specified by students with knowledge of the source code of *Cal*;
- Functional Testing using partition and boundary testing using commercial testers;
- Random Testing; and

[*]Supported by FAPESP: Process Number 98/16492-8

- Systematic Functional Testing.

We do not describe in detail any of the above except Systematic Functional Testing, which is a set of guidelines used in generating functional tests which attempts to ensure the best possible coverage of the input and output spaces of the programme.

When we compare these criteria against mutation testing we found that the test set generated by Systematic Functional Testing criterion killed all the non-equivalent mutants while other approaches scored significantly less than this. The full details are given below.

The rest of this paper is organized as follows. In Section 2 we describe Systematic Functional Testing criterion. In Section 3 we describe mutation testing and its application as a measure of the ability of a test set to expose errors in the programme. In Section 4 we present the results of our study. Finally, in Section 5, we highlight future work required to confirm our results.

2 Systematic Functional Testing

Functional testing regards a computer programme as a function and selects values for the input domain which should produce values in the output domain which are the correct ones. If the output values are correct, then the function which just been executed is the function which was specified, i.e. the programme is correct or is a programme which has identical behaviour for the given input data. The selection of test case to be input to a functional test is determined on the basis of the functions to be performed by the software. Additions to the approach include Equivalence Class Partitioning and Boundary Value Analysis that attempt to add some structure to this approach.

As defined by Roper [10], the idea behind Equivalence Class Partitioning is to divide the input and output domain into equivalence partitions or classes of data which, according to the specification, are treated identically. Therefore, any datum chosen from an equivalence class is as good as any other since it should be processed in a similar fashion. On the other hand, Boundary Value Analysis is also based on equivalence partitioning but it focuses on the boundaries of an partition to obtain the corresponding input datum that will represent such a partition.

Systematic Functional Testing attempts to combine these functional testing criteria such that, once the input and output domain have been partitioned, Systematic Functional Testing requires at least two test case of each partition to minimize the problem of co-incident errors masking faults. Systematic Functional Testing also requires the evaluation at and around the boundaries of each partition, and provides a set of guidelines, described in Section 2.1, to facilitate the identification of such test cases. To illustrate how to generate a test set using Systematic Functional Testing criterion, the *Cal* UNIX programme, that will be used as example in the remaining of this paper, is described in Section 2.2.

One strength of functional testing criteria, including Systematic Functional Testing, is that they require only the product specification to derive the testing requirements. In this way, it can be applied indistinctly to any software program (procedural or object-oriented) or software component, since no source code is required. On the other hand, as highlighted by Roper [10], because functional criteria are only based on the specification, they cannot assure that essential/critical parts of the implementation have being covered. For example, considering Equivalence Class Partitioning, although the specification may suggest that a group of data is processing identically, this may not in fact be the case. This serve to reinforce the argument that functional testing criteria and structural testing criteria should be used in conjunction. Moreover, it would also be beneficial if a test set generated based on functional criterion provides a high coverage of the implementation according to a given structural criterion. Systematic Functional Testing aims at fulfill this expectation.

2.1 Systematic Functional Testing Guidelines

The following guidelines show what type of data should be selected for various types of functions, input and output domains. Each guideline may lead to select one or more test cases, depending on whether it is applicable to the programme under testing.

Numeric Values

For the input domain of a function which computes a value based on a numeric input value, the following test case should be selected:

- Discrete values: test each one;
- Range of values: test endpoints and one interior value for each range.

For the output domain, select input values which will result in the values being generated by the software. The types of value output may or may not correspond to the same type of input; for example, distinct values input may produce a range of output values depending on other factors, or a range of input values may produce only one or two output values such as true or false. Choose values to appear in the output as follows:

- Discrete values: generate each one;
- Range of values: generate each endpoint and at least one interior value for each range.

Different Types of Value and Special Cases

Different types of value should also be both input and generated on output, as for example a blank space can be regarded as a zero in a numeric field.

Special cases such as zero should also always be selected individually, even if they are inside a range of values. Values on “bit boundaries” should be selected if values are packed into limited bit fields when stored, to ensure that they are both stored and retrieved correctly.

Illegal Values

Values which are illegal input should be included in the test case, to ensure that the software correctly rejects them. It should also be attempted to generate illegal output values (which should not succeed).

It is particularly important to select values just outside any numeric range. Selecting both the minimum value which is legal and the next lowest value will test that the software handles the bottom of a range of values correctly, and the maximum and next highest will check the top of a range of values.

Real Numbers

There are special problems when testing involves real numbers rather than integer values, since the accuracy stored will normally be different to the value entered. Real values are usually entered as powers of 10, stored as powers of 2, and then output as powers of 10 again. The boundary checking for real numbers cannot be exact, therefore, but should still be included in the test case. An acceptable range of accuracy error should be defined, with boundary values differing by more than that amount in order to be considered as distinct input values. In addition, very small real numbers should be selected and zero.

Variable Range

Special care needs to be taken when the range of one variable depends on the value of another variable. For example, suppose the value of a variable x can be anything from zero to whatever value the variable y has, and suppose y can be any positive value. Then the following cases should be selected for inclusion in the input test case:

$$\begin{array}{ccccc} x & = & y & = & 0 \\ x & = & 0 & < & y \\ 0 & < & x & = & y \\ 0 & < & x & < & y \end{array}$$

In addition, the following illegal values should also be selected:

$$\begin{array}{ccccc} y & = & 0 & < & x \\ 0 & < & y & < & x \\ & x & < & 0 & \\ & y & < & 0 & \end{array}$$

Arrays

When dealing with arrays in either input or output, there is the problem of variable array size, as well as variable data, to be considered. Just as with any value input or output, each element of an array should be tested for the values as listed above. In addition, the array size itself should be tested for minimum, maximum and intermediate sizes, not just for one but for all dimensions, and in all combinations.

It may be possible to simplify the testing somewhat by taking advantage of possible sub-structures; for example a row or column may be regarded as one unit for subsequent testing. The array should be tested first as a single structure, then as a collection of substructures, and each substructure should be tested independently.

Text or String Data

Text or string data (varying number of characters input as a logical group) needs to be checked for varying length (including no characters at all), and for the validity of each character. Sometimes only alphabetic characters are valid, and other times alphanumeric or some punctuation characters would be valid.

In addition, Systematic Functional Testing insists that at least two values are chosen from a given partition of the data. The reasoning being that if a single value is chosen to fit a partition co-incidental error may mask a fault, i.e. to a programme which is meant to take a number and output the square of that number then the input value of 2, expected output of 4 is not sufficient to distinguish between $2 * 2$ and $2 + 2$. Using a second value from a partition minimizes this problem.

2.2 Example Programme

An example may help to make clear the actual process of defining the domains for input and output, and how the domains affect the choice of test case. We are using the *Cal* programme in our example. The specification of *Cal* programme, extracted from UNIX man pages, is presented in Figure 1. Observe that *Cal* is a command line programme that can be called in three different ways: with no parameters, with one parameter, or with two parameters. In the first case it shows the calendar of the current month; in the second case it shows the calendar of a complete year (the specified one); and in the latter case it shows the calendar of a single month of a given year.

Considering z the number of the parameters, we can define two domains:

```
NAME
     cal - display a calendar

SYNOPSIS
     cal [ [ month ] year ]

AVAILABILITY
     SUNWesu

DESCRIPTION
     The cal utility writes a Gregorian calendar to standard out-
     put.   If the year operand is specified, a calendar for that
     year is written.  If no operands are specified,  a  calendar
     for the current month is written.

OPERANDS
     The following operands are supported:

     month   Specify the month to be displayed, represented as  a
             decimal  integer  from 1 (January) to 12 (December).
             The default is the current month.

     year    Specify  the  year  for  which   the   calendar   is
             displayed,  represented  as a decimal integer from 1
             to 9999.  The default is the current year.

ENVIRONMENT
     See environ(5) for descriptions of the following environment
     variables  that  affect  the  execution  of  cal:  LC_TIME,
     LC_MESSAGES, and NLSPATH.

EXIT STATUS
     The following exit values are returned:
     0          Successful completion.
     >0         An error occurred.

SEE ALSO
     calendar(1), environ(5)

NOTES
     An unusual calendar is printed for September 1752.  That  is
     the  month  11 days were skipped to make up for lack of leap
     year adjustments.  To see this calendar, type:
          cal 9 1752

     The command cal 83 refers to the year 83, not 1983.

     The year is always considered to start in January.
```

Figure 1: *Cal* man page.

- valid domain: $0 \leq z \leq 2$; and
- invalid domain: $z > 2$.

Observe that it is not possible to call a programme with a negative number of arguments. Therefore, the invalid domain $z < 0$ is not considered in this case. Table 1 (a) illustrates the partitions corresponding to the number of parameters. The number between parentheses identifies a single partition and is used to associate which test case is generated with respect to each partition.

Now, considering the case where the *Cal* programme is called with one argument (that represents a given year $yyyy$ from 1 to 9999), the valid and invalid domains are:

- invalid domain $yyyy < 1$;
- invalid domain $yyyy > 9999$;
- valid domain $1 \leq yyyy \leq 9999$.

Considering the case where the *Cal* programme is called with two arguments (a month *mm* of a given given year *yyyy*), the valid and invalid domains are:

- invalid domain $mm < 1$ and/or $yyyy < 1$;
- invalid domain $mm > 12$ and/or $yyyy > 9999$;
- valid domain $1 \leq mm \leq 12$ and $1 \leq yyyy \leq 9999$.

Observe that when the programme is called with two parameters, if one of than is invalid the equivalency class will be invalid. Table 1 (b) and Table 1 (c) summarize the equivalence classes for *Cal* programme considering one parameter and two parameters inputs, respectively.

Table 1: *Cal* Equivalence Partitioning Classes – Valid (V) and Invalid (I): (a) – number of parameters, (b) – one parameter input, (c) – two parameters input, (d) – one parameter output, and (e) – two parameters output.

Parameters	*$0 \leq z \leq 2$*	*$z > 2$*
z	V (1)	I (2)

(a)

Year	*non-integer*	*$yyyy < 1$*	*$yyyy > 9999$*	$1 \leq yyyy \leq 9999$
yyyy	I(3)	I(4)	I(5)	V(6)

(b)

Month/Year	*non-integer*	$yyyy < 1$	$yyyy > 9999$	$1 \leq yyyy \leq 9999$
non-integer	I (7)	I (8)	I (9)	I (10)
$mm < 1$	I (11)	I (12)	I (13)	I (14)
$mm > 12$	I (15)	I (16)	I (17)	I (18)
$1 \leq mm \leq 12$	I (19)	I (20)	I (21)	V (22)

(c)

Year	*Number of days*
1752	356 (23)
Any non-leap year	365 (24)
Any leap year	366 (25)

(d)

Month e Year	*Number of Days*
01,03,05,07,08,10,12/any year	31 (26)
04,06,09,11/any year	30 (27)
02/non-leap year	28 (28)
02/leap year	29 (29)
09/1752	20 (30)

(e)

Considering the output domain for *Cal* programme, it consists of the calendar of a single month or of an entire year. An error message is output if a invalid month and/or an invalid year is entered. Table 1 (d) summarizes the output classes to be considered based on the calendar of an entire year, and Table 1 (e) shows the output classes to be considered based on the calendar of a single month.

Once the partitions have been determined, considering the input and output domains, test cases are chosen to cover such partitions. First of all, values should be chosen covering the invalida partitions, at least one from each of the invalid domains. Next a few values should be chosen which lie well within the valid ranges.

Since the most fruitful source of good test cases are the boundaries of the domains, months 0, 1, 12 and 13 should be selected to ensure that 0 and 13 are invalid and 1 and 12 are valid. Similarly, years 0, 1, 9999, and 10000 should be selected. Negative values should also be input for month and year.

Considering the equivalence partitions of Table 1 and the guidelines described in Section 2.1, Table 2 contains the complete test set generated based on the Systematic Functional Testing criterion. In all, 76 test cases are generated to cover all the equivalence partitions. For example, TC_1 is a valid test case generated to cover the partitions 1 (valid partition for two parameters), 22 (valid month and valid year), and 30 (valid output of a single month with 20 days). On the other hand, TC_{33} is an invalid test case generated to cover the partitions 1 and 14 (invalid month and valid year).

Table 2: A complete pool of test cases for the *Cal* programme.

Test Case ID	**Input Parameters**	**Covered Partition**	**Is Valid**
TC_1	`9 1752`	1, 22, 30	Yes
TC_2	`2 1200`	1, 22, 29	Yes
TC_3	`2 1000`	1, 22, 29	Yes
TC_4	`2 1900`	1, 22, 28	Yes
TC_5	`2 1104`	1, 22, 29	Yes
TC_6	`2 2000`	1, 22, 29	Yes
TC_7		1	Yes
TC_8	`1`	1, 6, 24	Yes
TC_9	`1999`	1, 6, 24	Yes
TC_{10}	`7999`	1, 6, 24	Yes
TC_{11}	`1 1`	1, 22, 26	Yes
TC_{12}	`1 1999`	1, 22, 26	Yes
TC_{13}	`1 7999`	1, 22, 26	Yes
TC_{14}	`1 9999`	1, 22, 26	Yes
TC_{15}	`12 1999`	1, 22, 26	Yes
TC_{16}	`12 1`	1, 22, 26	Yes
TC_{17}	`12 7999`	1, 22, 26	Yes
TC_{18}	`12 9999`	1, 22, 26	Yes
TC_{19}	`6 1`	1, 22, 27	Yes
TC_{20}	`6 1999`	1, 22, 27	Yes
TC_{21}	`6 7999`	1, 22, 27	Yes
TC_{22}	`6 9999`	1, 22, 27	Yes
TC_{23}	`9 1`	1, 22, 27	Yes
TC_{24}	`9 1999`	1, 22, 27	Yes
TC_{25}	`9 7999`	1, 22, 27	Yes
TC_{26}	`9 9999`	1, 22, 27	Yes
TC_{27}	`8 1752`	1, 22, 26	Yes
TC_{28}	`10 1752`	1, 22, 26	Yes
TC_{29}	`9 1751`	1, 22, 27	Yes
TC_{30}	`9 1753`	1, 22, 27	Yes
TC_{31}	`2 1752`	1, 22, 29	Yes
TC_{32}	`0 2000`	1, 14	No
TC_{33}	`-1 2000`	1, 14	No
TC_{34}	`-14 2000`	1, 14	No
TC_{35}	`-12 2000`	1, 14	No
TC_{36}	`13 2000`	1, 18	No
TC_{37}	`3 0`	1, 20	No
TC_{38}	`3 -1`	1, 20	No
TC_{39}	`3 -9999`	1, 20	No
TC_{40}	`3 -10000`	1, 20	No
TC_{41}	`3 10000`	1, 21	No
TC_{42}	`a 2000`	1, 10	No
TC_{43}	`1.0 2000`	1, 10	Yes
TC_{44}	`3 z`	1, 19	No
TC_{45}	`3 2.0`	1, 19	No
TC_{46}	`10 1000 5`	2	No
TC_{47}	`+10 1000`	1, 22, 26	Yes
TC_{48}	`’(10)’ 1000`	1, 10	No
TC_{49}	`10 +1000`	1, 22, 26	Yes
TC_{50}	`10 ’(1000)’`	1, 19	No
TC_{51}	`0012 2000`	1, 22, 26	Yes
TC_{52}	`012 2000`	1, 22, 26	Yes
TC_{53}	`10 0083`	1, 22, 26	Yes
TC_{54}	`10 083`	1, 22, 26	Yes
TC_{55}	`10 2000 A`	2	No
TC_{56}	`10 A 2000`	2	No
TC_{57}	`A 10 2000`	2	No
TC_{58}	`2.0 10 2000`	2	No
TC_{59}	`10 2.0 2000`	2	No
TC_{60}	`10 2000 2.0`	2	No
TC_{61}	`9999`	1, 6, 24	Yes
TC_{62}	`0`	1, 4	No
TC_{63}	`10000`	1, 5	No
TC_{64}	`-9999`	1, 4	No
TC_{65}	`a`	1, 3	No
TC_{66}	`A b`	1, 7	No
TC_{67}	`a -1`	1, 8	No
TC_{68}	`a 10000`	1, 9	No
TC_{69}	`-1 a`	1, 11	No
TC_{70}	`-1 -1`	1, 12	No
TC_{71}	`-1 10000`	1, 13	No
TC_{72}	`13 a`	1, 15	No
TC_{73}	`13 -1`	1, 16	No
TC_{74}	`13 10000`	1, 17	No
TC_{75}	`1752`	1, 6, 23	Yes
TC_{76}	`2000`	1, 6, 25	Yes

3 An Overview of Mutation Testing

Mutation testing is a fault-based testing adequacy criterion proposed by DeMillo *et al.* [5]. Given a programme P, a set of alternative programmes M, called **mutants** of P, is considered in order to measure the adequacy of a test set T. The mutants differ from P only on simple syntactic changes, determined by a set of **mutant operators**. In fact, mutant operators can be seen as the implementation of a **fault model** that represents the common errors committed during software development. One example of such mutant operator in C is the replacement of the relational operator `<` in the code `if (a < b)` by each of the other relational operators `>`, `<=`, `>=`, `==` and `!=`.

To assess the adequacy of a test set T, each mutant $m \in M$, as well as the programme P, has to be executed against each the test case $t \in T$. If the observed output of a mutant m is the same as that of P for all test cases in T, then m is considered live, otherwise it is considered dead or eliminated. A live mutant m can be equivalent to programme P. An equivalent mutant can not be distinguished and is discarded from the mutant set as it does not contribute to improve the quality T.

The **mutation score** – the ratio of the number of dead mutants to the number of non-equivalent mutants – provides to the tester a mechanism to assess the quality of the testing activity. When a mutation score reaches 1.00, it is said that T is adequate with respect to (w.r.t.) mutation testing (**MT-adequate**) to test P.

Since the set of mutant operators can be seen as an implementation of a fault model, we can consider all the mutant operators as a set of faults against which our test sets is being evaluated. In this sense, a test set that kills all the mutants or almost all of them, can be considered effective in detecting these kind of faults.

In the case study described in this article the complete set of mutant operators for unit testing implemented in *PROTEUM/IM* 2.0 testing tool [4] is used as a fault model to evaluate the effectiveness of the Systematic Functional Testing and others approaches in detecting faults. Below we describe the case study carried out using the *Cal* programme and the results obtained.

4 Study Procedure and Results

The methodology used to conduct this case study comprises five steps: Programme Selection, Tool Selection, Test Set Generation, Results and Data Analysis.

4.1 Programme Selection

In this case study the *Cal* programme – an UNIX utility to show calendars – is used for both: (1) illustrate the process of generating test cases using Systematic Functional Testing criteria; and (2) also to make some comparisons with other functional test sets. The results obtained herein must be further investigated for larger programmes and other application domains.

4.2 Tool Selection

To support the application of Mutation Testing, *PROTEUM/IM* 2.0 [4] was used. This tool was developed at the *Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo* – Brazil. Some facilities that ease the carrying out of empirical studies are provided, such as:

- Test case handling: execution, inclusion/exclusion and enabling/disabling of test cases;
- Mutant handling: creation, selection, execution, and analysis of mutants; and
- Adequacy analysis: mutation score and statistical reports.

PROTEUM/IM 2.0 supports the application of mutation testing at the unit and integration level for C programmes. At unit level it implements a set of 75 mutant operators, divided into four groups according to where the mutation is applied: Constants (3 operators), Operators (46 operators), Statements (15 operators) and Variables (11 operators). At integration level 33 mutants operators are implemented. Given a connection between units f and g (f calls g), there are two groups of mutations: Group-I (24 operators) that applies changes to the body of function g; and Group-II (9 operators) that applies mutations to the places unit f calls g. More detailed information about *PROTEUM/IM* 2.0 testing environment can be found in [3].

In this paper the unit mutant operators were used as a fault model against which the test sets were evaluated. The complete set of unit mutant operators available in *PROTEUM/IM* 2.0 is presented in Appendix A.

Mutation testing has been found to be powerful in its fault detection capability when compared to other code coverage criteria at the unit and integration level [3, 8, 11, 12]. Although powerful, mutation testing is computationally expensive [3, 8, 11, 12]. Its high cost of application, mainly due to the high number of mutants created and the effort to determine the equivalent mutants, has motivated the proposition of many alternative criteria for its application [1, 2, 6–9].

4.3 Test Set Generation

The idea of this experiment is to evaluate the adequacy of functional and random test sets w.r.t. mutation testing. Therefore, different test sets were generated and their ability to kill mutants was evaluated.

One test set, named TS_{SFT}, was generated using the Systematic Functional Testing described in Section 2. Four test sets, named TS_{PB_1}, TS_{PB_2}, TS_{PB_3}, and TS_{PB_4}, were generated by students using both Equivalent Class Partitioning and Boundary Value Analysis criteria. And seven random test sets, named TS_{RA_1}, TS_{RA_2}, TS_{RA_3}, TS_{RA_4}, TS_{RA_5}, TS_{RA_6}, TS_{RA_7}, where generated containing 10, 20, 30, 40, 50, 60, and 70 test cases, respectively. In all, 12 test sets were generated and the cardinality of each test set is shown in Table 3. The second column of Table 3 presents the number of test cases in each test set. The third column presents the number of effective test case, i.e., test case that kills at least one mutant considering the order of execution. For example, considering TS_{SFT}, 76 test cases were generated to cover all valid and invalid partitions and, from this 76 test cases, 21 killed at least one mutant when executed.

Table 3: Functional and Random Test Sets.

Test Set	Number of test case	Effective test case
TS_{SFT}	76	21
TS_{PB_1}	21	17
TS_{PB_2}	15	13
TS_{PB_3}	21	17
TS_{PB_4}	14	13
TS_{RA_1}	10	5
TS_{RA_2}	20	9
TS_{RA_3}	30	16
TS_{RA_4}	40	22
TS_{RA_5}	50	23
TS_{RA_6}	60	27
TS_{RA_7}	70	29

4.4 Results and Data Analysis

To illustrate the cost aspect related to mutation testing, consider the *Cal* programme that has 119 LOC, 4,624 mutants were generated by the set of unit mutant operators implemented in *PROTEUM/IM* 2.0. In order to evaluate the coverage of a given test set against mutation testing its is necessary to determine the equivalent mutants. This activity was carried out by hand and 335 (7.24%) out of 4,624 generated mutants were identified as equivalents.

Having determined the equivalent mutants, we evaluated the mutation score obtained by each test set, i.e., we evaluated the ability of each test set to distinguish the faults modelled by the set of non-equivalent mutants. Table 4 shows, for each test set, the number of live mutants (i.e., the number of mutants that the test set was not able to detect), the percentage of live mutants with respect to the total number of generated mutants, the mutation score obtained, and the live mutants grouped by mutant operator class.

For example, it can be observed that TS_{SFT} is the only test set that revealed all faults modelled by the mutant operators. After its execution all the non-equivalent mutants are killed and a mutation score of 1.00 is obtained. Considering the test set TS_{PB_2}, after evaluating it w.r.t. mutation testing, 74 mutants are still alive, i.e., 1.60% of the total, and the mutation score obtained is around 0.983. The last columns show the number of live mutants per mutant operator class, giving an indication of the types of faults missing by the corresponding test set. For example, considering TS$_{PB_2}$, 33 out of 74 live mutants are from the Constant mutant operator class, 22 out of 74 are from the Operator class, and 19 out of 74 are from the Variable class.

Table 4: Test Set Coverage and Mutant Operator Class Missed.

Test Set	**Number of Live**	**Percentage of Live**	**Mutation Score**	**Missing Mutants per Class**			
				Constant	**Operator**	**Statement**	**Variable**
TS$_{SFT}$	0	0	1.000000	0	0	0	0
TS$_{PB_1}$	371	8.02	0.913500	193	78	27	73
TS$_{PB_2}$	74	1.60	0.982747	33	22	0	19
TS$_{PB_3}$	124	2.68	0.971089	58	31	13	22
TS$_{PB_4}$	293	6.34	0.931686	116	84	16	77
TS$_{RA_1}$	1,875	40.55	0.563242	944	539	103	289
TS$_{RA_2}$	558	12.07	0.870021	287	161	21	89
TS$_{RA_3}$	419	9.06	0.902399	216	113	15	75
TS$_{RA_4}$	348	7.53	0.918938	181	87	12	68
TS$_{RA_5}$	311	6.73	0.927557	159	77	11	64
TS$_{RA_6}$	296	6.40	0.931051	149	73	11	63
TS$_{RA_7}$	69	1.49	0.983927	21	30	0	18

A more detail information about the live mutants is presented in Table 5. In this table the live mutants are grouped per mutant operator. For example, considering TS$_{PB_2}$, the 33 live mutants from Constant class correspond to 17 mutants of u-Cccr, 13 of u-Ccsr, and 3 of u-CRCR. From Table 5 we can clearly observe that TS$_{SFT}$ is the only test set that revealed all the faults modelled by the set mutants. We believe that this occurs because TS$_{SFT}$ is designed to cover at least two test cases per partition to avoid co-incidental errors, what is not required by the other approaches. The other functional approaches, although having a lower application cost because they required less test cases, did not obtain a significative mutation score.

According to Table 4 it can be observed that TS$_{SFT}$ reaches the maximum coverage w.r.t. mutation testing. Only two other test sets reached a mutation score over 0.98 but lower than 1.00: TS$_{PB_2}$ and TS$_{RA_7}$. On average, considering random test sets with 10, 20, and 30 test cases, it can be observed that all the test sets generated based on function testing criteria scored over 0.91 while TS$_{RA_1}$, TS$_{RA_2}$, and TS$_{RA_3}$, determined mutation score around 0.56, 0.87, and 0.90, respectively.

Considering that the test set obtained by using Systematic Functional Testing criterion has 76 test cases and only 21 out of 76 are effective, considering the order of application, we observe that the random test sets with 70 an 20 test cases, scored relatively less than TS$_{SFT}$. TS$_{RA_7}$ determines an mutation score around 0.984 and TS$_{RA_2}$ determines an mutation score around 0.870, which represent scores 1.6% and 13% below the one determined by TS$_{SFT}$, respectively.

Considering only the TS$_{SFT}$ test set, as described before, we observed that, due to the order of execution, some test cases does not contribute to increment the mutation score, i.e., even if such test cases were removed from TS$_{SFT}$, the test set is still adequate w.r.t. mutation testing. From this evaluation we found that only 21 out of 76 test cases are effective. Table 6 shows the set of 21 test cases and also the increment in the mutation score produced by each one. Observe that a mutation score of 1.00 is obtained with this subset of test cases. For example, TC$_1$ has been executed, 2,097 mutants are still alive (45.35% w.r.t. the total of generated mutants) and a mutation score of 0.511 is obtained.

We carried out another analysis in TS$_{SFT}$ to identify which one of these 21 test cases are indispensable to obtain a mutation score of 1.00. By analyzing which test case killed each mutant, we observed that some mutants are killed by only one specific test case such that, if this particular test case is removed from the test set, such a test set is not adequate any more w.r.t. mutation testing, i.e., at least one mutant will remain alive. We called this test case as indispensable in the sense that, considering these particular test set, it is not possible to obtain a mutation score of 1.00 if one of such indispensable test cases is removed.

We found that 9 out of the 21 effective test test cases of TS$_{SFT}$ are indispensable and cannot be removed from the test set if a mutation score of 1.00 is required because there are some mutants that are killed only by one of these 9 test cases. We evaluate the mutation score that these 9 test cases determined w.r.t. mutation testing. The results are summarized in Table 7. As can be observed, the mutation score obtained by these 9 test cases is 0.983, the same mutation score determined by TS$_{PB_2}$ and TS$_{RA_7}$. Comparing with the random test sets TS$_{RA_1}$ (that has 5 out of 10 effective test cases) and TS$_{RA_2}$ (that has 9 out of 20 effective test cases), the difference in the mutation score is around 42% and 11%, respectively. This may indicate that even selecting random test sets with the same number of effective test cases, the efficacy in detecting faults depends of other factors that, in this case, were not satisfied by the random test sets.

Table 5: Test Set Coverage and Type of Mutation Missed.

Test Set	**Missing Operators per Class**			
	Constant	**Operator**	**Statement**	**Variable**
TS$_{SFT}$	–	–	–	–
TS$_{PB_1}$	u-Cccr(95) u-Ccsr(74) u-CRCR(24)	u-OAAA(8) u-OABA(6) u-OAEA(2) u-OASA(4) u-OEAA(10) u-OEBA(7) u-OESA(6) u-OLAN(1) u-OLBN(1) u-OLLN(1) u-OLNG(1) u-OLRN(3) u-OLSN(2) u-ORAN(7) u-ORBN(3) u-ORLN(2) u-ORRN(10) u-ORSN(4)	u-SRSR(7) u-SSDL(9) u-SSWM(1) u-STRI(2) u-STRP(8)	u-VDTR(4) u-VGAR(10) u-VLSR(43) u-VTWD(16)
TS$_{PB_2}$	u-Cccr(17) u-Ccsr(13) u-CRCR(3)	u-OAAN(1) u-OABN(1) u-OEAA(5) u-OEBA(4) u-OESA(4) u-OLRN(1) u-OLSN(2) u-ORRN(2) u-ORSN(2)	–	u-VDTR(3) u-VLSR(10) u-VTWD(6)
TS$_{PB_3}$	u-Cccr(28) u-Ccsr(24) u-CRCR(6)	u-OABN(1) u-OEAA(6) u-OEBA(4) u-OESA(4) u-OLAN(1) u-OLBN(1) u-OLLN(1) u-OLRN(2) u-OLSN(2) u-ORAN(2) u-ORBN(1) u-ORRN(4) u-ORSN(2)	u-SRSR(3) u-SSDL(4) u-SSWM(1) u-STRI(1) u-STRP(4)	u-VDTR(4) u-VGAR(1) u-VLSR(12) u-VTWD(5)
TS$_{PB_4}$	u-Cccr(48) u-Ccsr(53) u-CRCR(15)	u-OAAN(8) u-OABN(5) u-OALN(4) u-OARN(12) u-OASN(4) u-OEAA(19) u-OEBA(9) u-OESA(10) u-OLRN(1) u-OLSN(2) u-ORAN(2) u-ORRN(6) u-ORSN(2)	u-SRSR(5) u-SSDL(5) u-STRI(1) u-STRP(5)	u-VDTR(4) u-VLSR(44) u-VSCR(18) u-VTWD(11)
TS$_{RA_1}$	u-Cccr(480) u-Ccsr(334) u-CRCR(130)	u-OAAA(19) u-OAAN(43) u-OABA(14) u-OABN(32) u-OAEA(5) u-OALN(26) u-OARN(69) u-OASA(9) u-OASN(18) u-OCNG(4) u-OEAA(48) u-OEBA(23) u-OESA(24) u-Oido(2) u-OLAN(3) u-OLBN(1) u-OLLN(1) u-OLNG(6) u-OLRN(10) u-OLSN(6) u-ORAN(53) u-ORBN(32) u-ORLN(24) u-ORRN(43) u-ORSN(24)	u-SMTC(3) u-SMTT(3) u-SMVB(2) u-SRSR(27) u-SSDL(29) u-SSWM(2) u-STRI(7) u-STRP(29) u-SWDD(1)	u-VDTR(39) u-VGAR(31) u-VLAR(3) u-VLSR(161) u-VTWD(55)
TS$_{RA_2}$	u-Cccr(155) u-Ccsr(97) u-CRCR(35)	u-OAAA(10) u-OAAN(5) u-OABA(8) u-OABN(8) u-OAEA(3) u-OALN(6) u-OARN(7) u-OASA(5) u-OASN(2) u-OEAA(16) u-OEBA(9) u-OESA(8) u-OLNG(2) u-OLRN(5) u-OLSN(6) u-ORAN(16) u-ORBN(13) u-ORLN(5) u-ORRN(19) u-ORSN(8)	u-SRSR(6) u-SSDL(5) u-SSWM(2) u-STRI(2) u-STRP(6)	u-VDTR(5) u-VGAR(10) u-VLSR(52) u-VTWD(22)
TS$_{RA_3}$	u-Cccr(107) u-Ccsr(82) u-CRCR(27)	u-OAAA(8) u-OAAN(2) u-OABA(7) u-OABN(2) u-OAEA(2) u-OALN(4) u-OARN(6) u-OASA(5) u-OEAA(15) u-OEBA(8) u-OESA(8) u-OLNG(1) u-OLRN(4) u-OLSN(6) u-ORAN(8) u-ORBN(5) u-ORLN(2) u-ORRN(13) u-ORSN(7)	u-SRSR(4) u-SSDL(5) u-SSWM(1) u-STRI(1) u-STRP(4)	u-VDTR(2) u-VGAR(8) u-VLSR(45) u-VTWD(20)
TS$_{RA_4}$	u-Cccr(88) u-Ccsr(69) u-CRCR(24)	u-OAAA(8) u-OAAN(2) u-OABA(7) u-OABN(1) u-OAEA(2) u-OARN(2) u-OASA(5) u-OEAA(11) u-OEBA(7) u-OESA(6) u-OLNG(1) u-OLRN(3) u-OLSN(4) u-ORAN(7) u-ORBN(3) u-ORLN(2) u-ORRN(11) u-ORSN(5)	u-SRSR(3) u-SSDL(5) u-STRI(1) u-STRP(3)	u-VDTR(1) u-VGAR(8) u-VLSR(41) u-VTWD(18)
TS$_{RA_5}$	u-Cccr(66) u-Ccsr(69) u-CRCR(24)	u-OAAA(8) u-OAAN(2) u-OABA(7) u-OABN(1) u-OAEA(2) u-OARN(2) u-OASA(5) u-OEAA(6) u-OEBA(4) u-OESA(4) u-OLNG(1) u-OLRN(3) u-OLSN(4) u-ORAN(7) u-ORBN(3) u-ORLN(2) u-ORRN(11) u-ORSN(5)	u-SRSR(3) u-SSDL(4) u-STRI(1) u-STRP(3)	u-VDTR(1) u-VGAR(6) u-VLSR(39) u-VTWD(18)
TS$_{RA_6}$	u-Cccr(56) u-Ccsr(69) u-CRCR(24)	u-OAAA(8) u-OAAN(2) u-OABA(7) u-OABN(1) u-OAEA(2) u-OARN(2) u-OASA(5) u-OEAA(5) u-OEBA(4) u-OESA(4) u-OLNG(1) u-OLRN(3) u-OLSN(4) u-ORAN(6) u-ORBN(3) u-ORLN(2) u-ORRN(9) u-ORSN(5)	u-SRSR(3) u-SSDL(4) u-STRI(1) u-STRP(3)	u-VDTR(1) u-VGAR(6) u-VLSR(39) u-VTWD(17)
TS$_{RA_7}$	u-Cccr(5) u-Ccsr(13) u-CRCR(3)	u-OAAN(2) u-OABA(1) u-OABN(1) u-OARN(2) u-OASA(1) u-OEAA(5) u-OEBA(4) u-OESA(4) u-OLRN(1) u-OLSN(2) u-ORAN(2) u-ORRN(4) u-ORSN(1)	–	u-VLSR(9) u-VTWD(9)

Table 6: Effective Test Cases of TS_{SFT}: Mutation Score Increment.

Test Case	**# Live**	**% Live**	**Score**
TC_1	2,097	45.35	0.511075
TC_2	1,995	43.14	0.534857
TC_3	1,986	42.95	0.536955
TC_4	1,691	36.57	0.605736
TC_6	1,659	35.88	0.613197
TC_7	1,262	27.29	0.705759
TC_8	256	5.54	0.940312
TC_9	228	4.93	0.946841
TC_{11}	212	4.58	0.950571
TC_{14}	208	4.50	0.951504
TC_{15}	204	4.41	0.952436
TC_{32}	163	3.53	0.961996
TC_{33}	162	3.50	0.962229
TC_{36}	137	2.96	0.968058
TC_{37}	96	2.08	0.977617
TC_{38}	95	2.05	0.977850
TC_{41}	70	1.51	0.983679
TC_{61}	66	1.43	0.984612
TC_{62}	25	0.54	0.994171
TC_{63}	1	0.02	0.999767
TC_{64}	0	0.00	1.000000

Table 7: TS_{SFT}: Indispensable Test Cases.

Test Case	**# Live**	**% Live**	**Score**
TC_1	2,097	45.35	0.511075
TC_7	1,266	27.38	0.704826
TC_8	260	5.62	0.939380
TC_{36}	215	4.65	0.949872
TC_{41}	169	3.65	0.960597
TC_{61}	137	2.96	0.968058
TC_{62}	96	2.08	0.977617
TC_{63}	72	1.56	0.983213
TC_{64}	71	1.54	0.983446

Missing Operators
`u-Cccr(15) u-Ccsr(22) u-CRCR(6) u-OLRN(2) u-ORAN(4) u-ORBN(1) u-ORRN(6) u-SRSR(1) u-VDTR(4) u-VLSR(4) u-VTWD(6)`

5 Conclusions

From this study we can see the application of mutation testing as a coverage measure gives us assurance that the test set we have produced for a programme is effective in detecting faults and we would recommend doing this at least every time the method of test specification or programme production is changed.

Considering the *Cal* programme, we can see that the test set generated by Systematic Functional Testing killed 100% of the non-equivalent mutants, while the test sets generated based on other criteria applied to the same programme scored significantly less. We know that it is necessary to repeat the same experiment to a number of programmes and to see if the results from applying it to *Cal* are consistently repeated. To aid this it is the intention to place everything needed into a package suitable to allow such repetition to occur.

If Systematic Functional Testing demonstrates, in these repetition studies, to be as effective in detecting faults as in the case study presented in this paper, given the cost and effort involved in doing path level structural testing, Systematic Functional Testing can be a good start point to evaluate the quality of a software program/component since the criterion does not require the source code to be supplied. Due to the limitation of any functional testing criterion, an incremental testing strategy, combining Systematic Functional Testing with structural testing criteria, can be established taking the advantage of the strength of each testing technique.

References

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, Sept. 1979.

[2] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Towards the determination of sufficient mutant operators for C. In *First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000. (Special issue of the Software Testing Verification and Reliability Journal, 11(2), 2001 – To Appear).

[3] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, Mar. 2001.

[4] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Mutation 2000 Symposium*, pages 91–101, San Jose, CA, Oct. 2000. Kluwer Academic Publishers.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, Apr. 1978.

[6] A. P. Mathur. Performance, effectiveness and reliability issues in software testing. In *15th Annual International Computer Software and Applications Conference*, pages 604–605, Tokio, Japan, Sept. 1991. IEEE Computer Society Press.

[7] E. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *The Journal of Software Testing, Verification and Reliability*, 9(4):205–232, Dec. 1999.

[8] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, Apr. 1996.

[9] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *15th International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 1993. IEEE Computer Society Press.

[10] M. Roper. *Software Testing*. McGrall Hill, 1994.

[11] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, Dec. 1995.

[12] W. E. Wong, A. P. Mathur, and J. C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength, and effectiveness. In *International Conference on Software Quality and Productivity*, pages 258–265, Hong Kong, Dec. 1994. Chapman and Hall.

A Description of the Mutation Testing Unit Operators

PROTEUM/IM 2.0 has 75 unit mutation operators divided into 4 classes: Constant, Statement, Variable and Operator. The three first classes are presented in Table 8 and the Operator class is illustrated in Table 9

Table 8: Constant, Statement and Variable Classes Operators.

Constant	
Operator	Description
u-Cccr	Constant for Constant Replacement
u-Ccsr	Constant for Scalar Replacement
u-CRCR	Required Constant Replacement
Statement	
Operator	Description
u-SBRC	break Replacement by continue
u-SBRn	break Out to Nth Level
u-SCRB	continue Replacement by break
u-SCRn	continue Out to Nth Level
u-SDWD	do-while Replacement by while
u-SGLR	goto Label Replacement
u-SMTC	n-trip continue
u-SMTT	n-trip trap
u-SMVB	Move Brace Up and Down
u-SRSR	return Replacement
u-SSDL	Statement Deletion
u-SSWM	switch Statement Mutation
u-STRI	Trap on if Condition
u-STRP	Trap on Statement Execution
u-SWDD	while Replacement by do-while
Variable	
Operator	Description
u-VDTR	Domain Traps
u-VGAR	Mutate Global Array References
u-VGPR	Mutate Global Pointer References
u-VGSR	Mutate Global Scalar References
u-VGTR	Mutate Global Structure References
u-VLAR	Mutate Local Array References
u-VLPR	Mutate Local Pointer References
u-VLSR	Mutate Local Scalar References
u-VLTR	Mutate Local Structure References
u-VSCR	Stucture Component Replacement
u-VTWD	Twiddle Mutations

Table 9: Operator Class Operators.

Operator	Description
u-OAAA	Arithmetic Assignment Mutation
u-OAAN	Arithmetic Operator Mutation
u-OABA	Arithmetic Assignment by Bitwise Assignment
u-OABN	Arithmetic by Bitwise Operator
u-OAEA	Arithmetic Assignment by Plain Assignment
u-OALN	Arithmetic Operator by Logical Operator
u-OARN	Arithmetic Operator by Relational Operator
u-OASA	Arithmetic Assignment by Shift Assignment
u-OASN	Arithmetic Operator by Shift Operator
u-OBAA	Bitwise Assignment by Arithmetic Assignment
u-OBAN	Bitwise Operator by Arithmetic Assignment
u-OBBA	Bitwise Assignment Mutation
u-OBBN	Bitwise Operator Mutation
u-OBEA	Bitwise Assignment by Plain Assignment
u-OBLN	Bitwise Operator by Logical Operator
u-OBNG	Bitwise Negation
u-OBRN	Bitwise Operator by Relational Operator
u-OBSA	Bitwise Assignment by Shift Assignment
u-OBSN	Bitwise Operator by Shift Operator
u-OCNG	Logical Context Negation
u-OCOR	Cast Operator by Cast Operator
u-OEAA	Plain assignment by Arithmetic Assignment
u-OEBA	Plain assignment by Bitwise Assignment
u-OESA	Plain assignment by Shift Assignment
u-Oido	Increment/Decrement Mutation
u-OIPM	Indirection Operator Precedence Mutation
u-OLAN	Logical Operator by Arithmetic Operator
u-OLBN	Logical Operator by Bitwise Operator
u-OLLN	Logical Operator Mutation
u-OLNG	Logical Negation
u-OLRN	Logical Operator by Relational Operator
u-OLSN	Logical Operator by Shift Operator
u-ORAN	Relational Operator by Arithmetic Operator
u-ORBN	Relational Operator by Bitwise Operator
u-ORLN	Relational Operator by Logical Operator
u-ORRN	Relational Operator Mutation
u-ORSN	Relational Operator by Shift Operator
u-OSAA	Shift Assignment by Arithmetic Assignment
u-OSAN	Shift Operator by Arithmetic Operator
u-OSBA	Shift Assignment by Bitwise Assignment
u-OSBN	Shift Operator by Bitwise Operator
u-OSEA	Shift Assignment by Plain Assignment
u-OSLN	Shift Operator by Logical Operator
u-OSRN	Shift Operator by Relational Operator
u-OSSA	Shift Assignment Mutation
u-OSSN	Shift Operator Mutation