Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

INTERFACE MUTATION TEST ADEQUACY CRITERION: AN EMPIRICAL EVALUATION

MÁRCIO EDUARDO DELAMARO
JOSÉ CARLOS MALDONADO
ALBERTO PASQUINI
ADITYA P. MATHUR

Nº 83

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
Mar./1998

Abstract

In this paper we present an experiment to evaluate an inter-procedural test adequacy criterion named Interface Mutation. Program *SPACE*, developed for the European Space Agency (ESA), has been used in this experiment. It has a development record that keeps track of all the faults uncovered during its life cycle. Using this information we reproduced in the experiment a testing process, starting with a version of the *SPACE* with several faults and then applying Interface Mutation. In this way we could evaluate the cost and the fault revealing effectiveness of the Interface Mutation criterion. The results show that Interface Mutation is an effective, reasonable-cost criterion.

1 Introduction

Software testing is considered a phased, incremental activity performed during the software development cycle. Criteria and methods have been pursued aiming at the establishment of a low-cost and effective testing strategy. Evaluating the quality of a given test set is a key task in the testing activity. Coverage criteria, also known as test adequacy criteria, provide useful information to a decision maker. These criteria also provide information on how to improve a given test set.

Considerable research has gone into criteria definition, implementation of supporting tools and their evaluation. Among the available criteria, data flow [13] and mutation based criteria [5] have been found to be the most promising in terms of their error revealing effectiveness. These criteria have been well-studied at the unit testing level but few initiatives of using them at integration testing level can be identified. In principle, they could also be applied for the assessment of test sets for subsystems. However, such an application is likely to be inefficient due to program size, redundant due to the multiple test teams that might test the same part of the code, and incomplete because the large program size might prohibit testers from applying the criteria exhaustively. Notably, data-flow based criteria have been explored by Harrold and Soffa at the subsystem integration level [8]. Mutation-based criteria have been extended for application at the subsystem integration level,resulting in the criterion named Interface Mutation [1, 4]. Interface Mutation aims at the assessment of the adequacy of testing the interfaces between the units of a program, providing objective measurements on the adequacy of Integration Testing activity.

Despite the large number of mutant programs generated during the use of a mutation-based criterion, there is a strong motivation to effectively use the mutation technique at the integration level. This motivation stems from the earlier studies wherein mutation testing was found to be powerful in fault detection effectiveness when compared with other test adequacy criteria. Alternative approaches, such as Constrained Mutation [17], that may also be used at the integration level, have shown how the cost of testing, measured in terms of the number of mutants, can be kept within acceptable bounds without a significant reduction in its effectiveness.

The results of a pilot experiment [3] motivated further development to support the use and evaluation of Interface Mutation criterion. The development of a tool named *PROTEUM/IM* to support Interface Mutation has been carried out [1]. Supported by *PROTEUM/IM*, another study, using a Sort UNIX program, was conducted by introducing artificially created faults and evaluating the cost and effectiveness of Interface Mutation to reveal those faults [1, 4]. We believe that cost and effectiveness are the two significant factors against which a criterion should be evaluated and compared [7]. The results of our study are encouraging in that they indicate that the use of Interface Mutation criterion, especially in combination with cost reducing approaches, may constitute an effective and not too expensive way to evaluate and build test sets for Integration Testing.

These empirical studies led the authors to look for more meaningful and realistic programs and environments to apply and evaluate the use of Interface Mutation in software production. This paper reports a case study based on an application developed for the European Space Agency (ESA). This application has been used in many studies. An error log has kept track of the faults found and

corrections applied during its development and evolution, providing an experimentation scenario close to reality.

The next section presents our approach to integration testing based on Interface Mutation. Section 3 presents the methodology used in the experimental study conducted using Interface Mutation on the *SPACE* program. Section 4 presents and discusses the results of this experiment in terms of cost and fault revealing effectiveness. Section 5 provides our conclusions.

2 Interface Mutation: An Overview

The idea underlying mutation testing is to evaluate the quality of a test set based on its ability to reveal simple faults injected in the program being tested. It is expected that a test set capable of revealing simple faults is capable of revealing more complex faults. Experimental results have confirmed this expectation [10].

It has been shown that mutation testing is an effective criterion for test case selection and evaluation [14, 16, 18]. However, its cost, measured in terms of the number of mutants to be executed, is a barrier to be overcome to make it applicable to relatively large programs (e.g. of size over 100,000 lines of executable code). For unit testing, some approaches can be taken to reduce the cost of mutation testing, for example, by applying the constrained mutation criteria [17], without any significant loss in the effectiveness to reveal errors. To our knowledge research efforts on mutation testing [17, 9, 11, 5] have been concentrated at the unit testing level and dealt with programs of relative small size.

In earlier work [1, 4], the Interface Mutation criterion has been introduced aimed at making feasible the use of mutation testing during the integration testing phase. Interface Mutation aims at testing the interactions between units. The approach used in Interface Mutation is based on three key ideas: 1) restrict mutation operators to model integration errors; 2) test connections between two modules separately, one at a time; and 3) apply the Interface Mutation operators only on those parts of the modules that are related to module interactions such as function calls, parameters or global variables.

Integration errors are caused by incorrect values exchanged through the connection between modules. Thus, the "perturbations", or mutations, at this level are introduced directly and only at the objects responsible for the interaction between modules. In conventional languages like C, two modules are connected through function calls. In a call from a function F to a function G, there are four ways of data exchange which are not mutually exclusive. These are:

- Data can be passed to G via input parameters (parameters passed by value).
- Data can be passed to G and/or returned to F via input/output parameters (parameters passed by reference).
- Data can be passed to G and/or returned to F via global variables.
- Data can be returned to F via return values (as in *return* statements in C).

Using the coupling effect we argue that test cases able to distinguish the interface-mutants should be able to reveal integration errors. Obviously, the validity of this argument depends on the set of Interface Mutation operators.

One advantage of Interface Mutation is that, by design, it reduces the number of mutants to be analyzed. This is so because mutant operators are applied only to points in the program related to the connections between modules. It is possible for the tester to select one connection, to generate mutants related to that connection and to select test cases to distinguish those mutants, supporting incremental or non-incremental integration testing strategies.

Thus, the application of such operators must take into consideration the place from where a module is called and only apply the operator if this place is the desired one. This is equivalent to saying that the condition to distinguish a mutant at the level of Interface Mutation is changed in relation to ordinary

mutation testing. As stated by DeMillo [6], there are three conditions to distinguish a mutant **M** from a program **P** by a test case t: 1) reachability, the execution of **P** with t must execute to the point where the mutation is introduced; 2) necessity, the state of **M** must differ from the state of the original program immediately after the execution of the mutated statement; and 3) sufficiency, the difference must be propagated to a point such that different outputs are produced for **P** and **M**. In the context of Interface Mutation the reachability condition has changed. It now requires that the mutatnt must be executed through a path that includes the execution of a specific connection. For languages like C this kind of mutation requires a run time decision on whether the mutation should be applied or not [1].

Interface Mutation Operators for C

In order to precisely establish a mutation based criterion, it is necessary to define a set of mutation operators. Since the definition of mutation operators depends on the programming language, the Interface Mutation operator set presented here is for the C language.

Consider a program **P** and a test case t for **P**. Suppose that in **P** there are functions F and G such that F makes calls to G. Consider $S_I(G)$ to be the set of values passed to G and $S_O(G)$ the set of values returned by G. The definition of Interface Mutation operators is based on the following integration error types.

1. Type 1: Upon entering G, $S_I(G)$ does not have the expected values and these values lead to an erroneous output (a failure) produced before returning from G.

2. Type 2: Upon entering G, $S_I(G)$ does not have the expected values and these values cause $S_O(G)$ to assume wrong values that lead to an erroneous output (a failure) after returning from G.

3. Type 3: Upon entering G, $S_I(G)$ has the expected values but incorrect values in $S_O(G)$ are produced inside G and these incorrect values lead to an erroneous output (a failure) after returning from G.

Error Type 1 is, for example, to call to a function passing an incorrect parameter or with an incorrectly set global flag and this function produces an incorrect output. The flow in this case is shown in Figure 1a. In error Type 2 there is an incorrect value entering the module and another leaving the module and influencing an incorrect output after returning from G as shown in Figure 1b. This is the case, for instance, of a parameter of a function used to calculate a return value. An incorrect entering parameter may lead to an incorrect return value that may lead to an incorrect output. Type 3 has only a flow of incorrect values leaving the module. It is the case of a function called with correct parameters but carrying out an incorrect calculation that produces an incorrect return value leading to an incorrect output upon return from G. This situation is represented in Figure 1c.

This is a broad definition and does not specify the place of the fault which causes the error. It simply considers the existence of incorrect values entering or exiting a function call. This excludes, for example, the case when $S_I(G)$ has the expected values but a fault inside G produces an erroneous output before returning from G. In this case, there is no error propagation through the connection F-G. This type of error should have been detected at unit level.

In addition, more than one integration error can be associated with (or caused by) a single fault. For example, consider a program with three functions R, S and T such that R calls S and then T. Suppose that in function S an incorrect value $x \in S_O(S)$ is returned and this value is also part of $S_I(T)$. Suppose that due to x, T produces an incorrect output. Thus, a fault in S produced a Type 3 error in the connection R-S and a Type 1 error in the connection R-T.

Finally, the sets $S_I(G)$ and $S_O(G)$ depend in part on the program language. For example, for the C language they can be defined as:

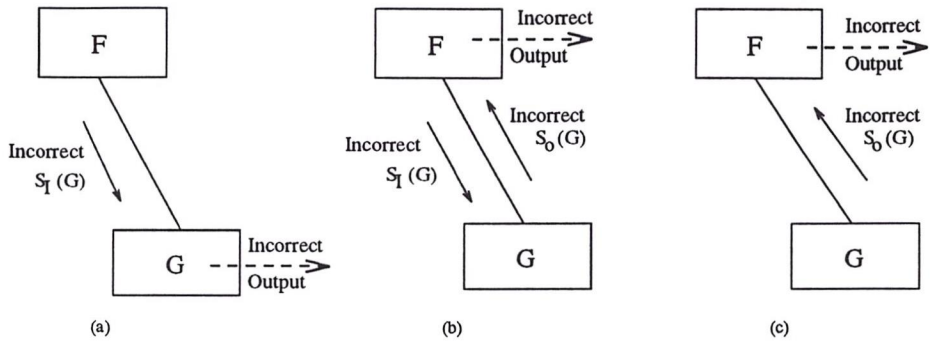

Figure 1: Type of Errors

- $S_I(G)$: The set of input values in a call to a function G is determined by the set of
  - input parameters used in the function call and
  - global variables used in G.
- $S_O(G)$: The set of output values in a call to a function G is determined by the set of
  - output parameters used in the function call
  - global variables used in G
  - values returned by G

Two groups of mutation operators are defined. Considering a connection between functions f-g, the first group (Group I) applies mutations inside the body of function g. This first group requires the preparation of the mutation at the point where f calls g such that the mutation is applied only if g is called through this point. Otherwise, function g behaves as in the original program. The second group (Group II) applies mutations to the call to g inside f.

Each operator has a name that identifies its semantics. In order to define the operators summarized in Tables 1 and 2, the following sets must be defined assuming that connection f-g is being tested:

$P(g)$: The set of formal parameters of g. This set also includes dereferences to pointer and array parameters. So, for example when a parameter v is defined "int *v", v belongs to this set as well as "$*v$".

$G(g)$: The set of global variables accessed by g.

$L(g)$: The set of variables defined in g (local variables).

$E(g)$: The set of global variables not used in g.

$C(g)$: The set of constants used in g.

The elements of sets P and G, respectively, are the formal parameters and the global variables used in g and referred to as "interface variables". The elements of the other sets are referred to as "non-interface variables and constants".

In addition, one more set R is defined but it does not depend on g. This set is the set of "required constants". It contains some special values, relevant for each data type and the associated operators. Table 3 summarizes these constants.

Table 1: List of Interface Mutation operators, Group I.

GROUP I	
Name	Meaning
DirVarRepPar	Replaces interface variable by each element of P
DirVarRepGlo	Replaces interface variable by each element of G
DirVarRepLoc	Replaces interface variable by each element of L
DirVarRepExt	Replaces interface variable by each element of E
DirVarRepCon	Replaces interface variable by each element of C
DirVarRepReq	Replaces interface variable by each element of R
IndVarRepPar	Replaces non interface variable by each element of P
IndVarRepGlo	Replaces non interface variable by each element of G
IndVarRepLoc	Replaces non interface variable by each element of L
IndVarRepExt	Replaces non interface variable by each element of E
IndVarRepCon	Replaces non interface variable by each element of C
IndVarRepReq	Replaces non interface variable by each element of R
DirVarIncDec	Inserts/removes increment and decrement operations at interface variable uses.
IndVarIncDec	Inserts/removes increment and decrement operations at non interface variable uses.
DirVarAriNeg	Inserts arithmetic negation at interface variable uses
IndVarAriNeg	Inserts arithmetic negation at non interface variable uses
DirVarLogNeg	Inserts logical negation at interface variable uses
IndVarLogNeg	Inserts logical negation at non interface variable uses
DirVarBitNeg	Inserts bit negation at interface variable uses
IndVarBitNeg	Inserts bit negation at non interface variable uses
RetStaDel	Deletes `return` statement
RetStaRep	Replaces `return` statement
CovAllNod	Coverage of all nodes
CovAllEdg	Coverage of all edges

Table 2: List of Interface Mutation operators, Group II.

GROUP II	
Name	Meaning
ArgRepReq	Replaces arguments by each element of R
ArgStcAli	Switches arguments of compatible type
ArgStcDif	Switches arguments of compatible type
ArgDel	Removes argument
ArgAriNeg	Inserts arithmetic negation at arguments
ArgLogNeg	Inserts logical negation at arguments
ArgBitNeg	Inserts bit negation at arguments
ArgIncDec	Argument Increment and Decrement
FunCalDel	Removes function call

The set of operators presented in Tables 1 and 2 has been implemented in a tool called *PROTEUM/IM*. This tool is an evolution of Proteum [2], a tool for applying mutation testing for C programs at unit level. In addition to the usual operations required from such tools, Proteum and *PROTEUM/IM* implement several features to ease their use in empirical studies. Among them, we can cite:

- Graphical interface and script-oriented interface;
- Parameterizations that allow the reduction in the number of mutants generated; and
- Implementation of alternative mutation criteria.

In the next section we describe the experiment using the *SPACE* program carried out with *PROTEUM/IM*.

Table 3: Required constant sets.

Variable Type	Required Constants
signed integer signed char signed long	-1, 1, 0, MAXINT, MININT
unsigned integer unsigned char unsigned long enum	-1, 1, 0, MAXUNSIGNED
float double	-1.0, 1.0, 0.0, -0.0

The constants MAXINT, MININT and MAXUNSIGNED correspond respectively to the largest positive integer, smallest negative integer and largest unsigned integer. These values are machine and data type dependent.

3 Methodology

The goal of this experiment is to evaluate the cost and fault revealing effectiveness of the Interface Mutation criterion. The question that captures this goal is: "*What is the cost and probability of revealing faults in a program through the application of Interface Mutation ?*"

The general structure of our experiment is described in Figure 2. The steps used are described in subsections 3.1 to 3.6.

```
1  select Program to Test (PTT)
2  FOR each experiment from 1 to n
3       generate a test case subdomain based on PTT's
        operational profile
4       determine a set N of faults to use in the experiment
5       DO
6          introduce set N into PTT
7          apply I.M. to the current faulty version
8          determine the set M of faults revealed by IM
9          N := N - M
10       WHILE ( M is not empty )
11 END
12 collect and analyze data
```

Figure 2: General structure of the experiments to evaluate Interface Mutation

3.1 Program Selection

The first step in the experiment is to choose the program where Interface Mutation is to be applied. In general, one of two approaches could be taken. One approach is to use a real program and real faults found during its development. Another approach is to use a case study program and "probable", artificial faults. This latter approach has been used to evaluate Interface Mutation in a case study [4] reported earlier. For the experiment described here we used the first of the two approaches. The *SPACE* program has been selected for this purpose. We believe that *SPACE* is a good choice because it is a program in use by the European Space Agency (ESA), it has a record that keeps track of its evolution across its life cycle and it is a program of moderate complexity, as shown in Table 4.

SPACE processes a high level language named *Array Definition Language*. This language allows

Table 4: Complexity metrics of the *SPACE* program

	LOC's	Connections
Average by function	32.95	3,29
Total (135 functions)	4449	444

the user to describe an array of antennas. Its scope is to produce as output a data file – in a predefined format – based on the high level description. With the Array Definition Language the user can describe a certain array of antennas by a few simple statements rather than having to write the complete list of element positions and excitations. This list is produced as output of the program execution. An example of input and output of *SPACE* appears in Appendix A.

SPACE has approximately 4500 executable lines of code. It is subdivied into three subsystems [12]: a Parser subsystem, a Computation subsystem, and a Formating subsystem. The main tasks of the Parser subsystem are:

- Perform syntactic analysis on input sentence.
- Setup the internal data structure describing the array.

The main tasks of the Computation subsystem are:

- Assign or compute array parameters when omitted by the user.
- Execute consistency checks on user specified parameters.
- Compute the position and excitation of each array element.

The main task of Formatting subsystem is:

- Format the output data in the required format.

During development, testing and maintenance of *SPACE* a total of 33 faults were detected, recorded, and removed. The final version used in our experiment, referred as $SPACE_{\Phi}$, is considered to be a "non-faulty" version and used as an oracle. These 33 faults are classified in Table 5 according to the IEEE Standard 1044 "Standard Classification for Software Errors, Faults and Failures"

Table 5: Classification of faults revealed in *SPACE*

FAULT TYPE	SUBTYPE	#
Logic omitted or incorrect	Forgotten cases or steps	2
	Unnecessary function	1
	Missing condition test	4
	Checking wrong variable	2
Computational problems	Equation insufficient or incorrect	10
Interface incorrect or incomplete	Module mismatch	3
Data handling problems	Data initialized incorrectly	1
	Data accessed or stored incorrectly	10

Steps 2 to 10 of Figure 2, described below, were repeated 10 times. Each of this repetitions was called an "Experiment" and is refreed to by Roman numerals I to X. Multiple experiments were conducted to reduce chances of false conclusions.

3.2 Subdomain Generation

The developers of *SPACE* have also defined an Operational Profile for the program. This operational profile allows the creation of test data sets that represent subdomains of the entire input domain.

The approach used to evaluate the effectiveness of Interface Mutation criterion considers only test cases that distinguish at least one live mutant. Previous results [15] support the use of an adequacy criterion as a filter to reduce the size of a test set. Wong et al. have shown that coverage is a more determinant factor of the fault detection effectiveness than the size of the test set. In the *SPACE* experiment we have used a set of 2000 test cases to represent a subdomain. Before running the experiments we had generated subdomains of 2000, 4000, 6000, 8000 and 10000 test cases (one subdomain of each size) and executed the mutants with all the test cases from each subdomain. The mutation scores obtained did not vary significantly with the different size subdomains. This implies that – using this particular operational profile – the growth in the subdomain size does not cause a significant improvement in the mutation score and hence we could use a smaller subdomain and obtain almost the same result in terms of distinguishing mutants.

3.3 Fault Selection

Using such subdomains as representative of the entire input domain we can obtain a measure of how hard it is to reveal each of the known faults of *SPACE*. Thus, Step 4 in Figure 2 consists of creating several versions of the *SPACE* program, each of them containing exactly one fault, and executing each version with each element of the subdomain created in Step 3. Thus we compute, for each fault, a measure called "Hardness Index" (HI), given by the percentage of test cases that reveal the presence of the fault. Based on this measure, we selected for use in our experiments only those faults that are intuitively harder to reveal. Faults with HI lower than 10% were selected. Our goal was to avoid those faults that would certainly be revealed by any test set, independent of the test adequacy criterion used.

Steps 5 to 10 of Figure 2 are described next. These steps intend to reproduce the steps of program testing. We assume that during program development testing is initiated with a version of a program containing one or more faults. Through the application of a test selection criterion, test cases are derived and the program executed on these until a fault is revealed or until an adequacy criterion is satisfied. If the presence of a fault is revealed, the program is corrected and testing resumed.

In our experiment, $SPACE_{\Phi}$ version is taken and some of the known faults (set N) reinserted in order to create an incorrect version of the program referred as $SPACE_N$. On this version, Interface Mutation is applied to select test cases. If a selected test case t reveals an error in the program, i.e. $SPACE_N$ behaves differently than $SPACE_{\Phi}$, then one or more faults are chosen (the set M) and removed such that the new version behaves correctly on t. In other words, faults are removed such that $SPACE_{N-M}(t) = \dot{S}PACE_{\Phi}(t)$. This process is repeated until no fault can be revealed either because there are no faults to be revealed (version $SPACE_{\Phi}$ is reached) or because the test cases selected by Interface Mutation are not able to reveal the existing faults.

3.4 Fault Insertion

Once the set N of faults has been defined, the faults are re-inserted into $SPACE_{\Phi}$ by a mechanism embedded in the program's source code. Through conditional compilation it is possible to select which faults to insert. Faults numbered from 1 to 33 are inserted through a single compilation that "activates" some faults and "deactivates" others depending on an input parameter to the compilation process.

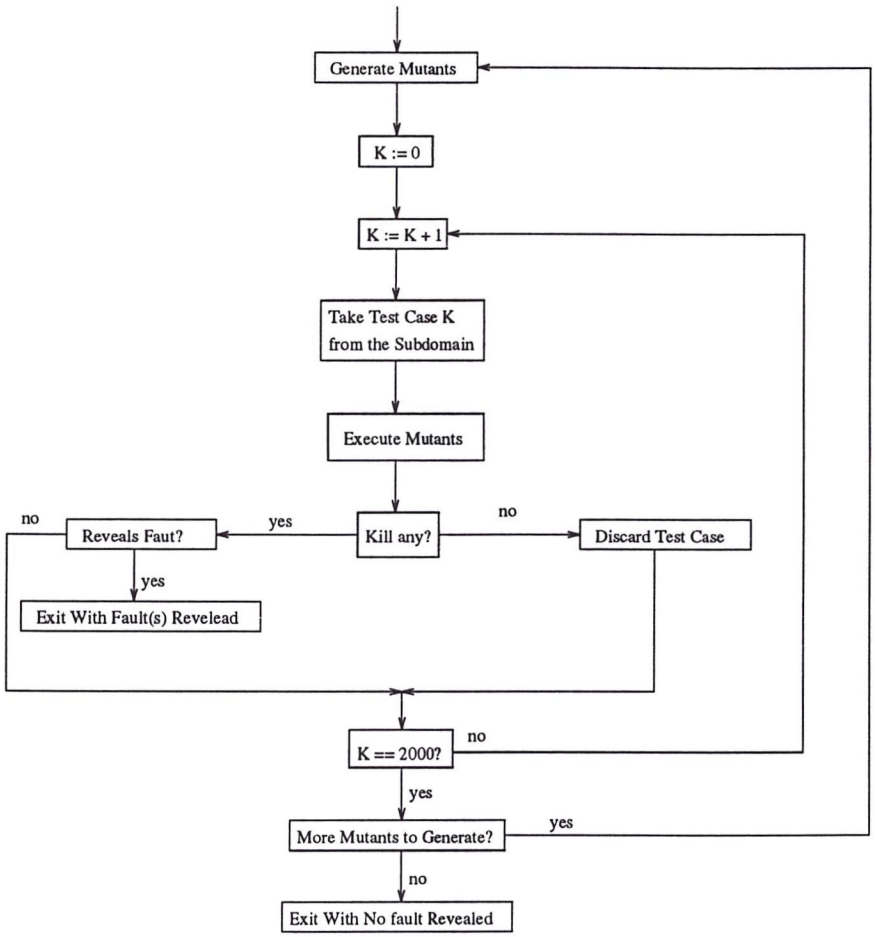

Figure 3: Test case selection process for each of the experiment

3.5 Application of Interface Mutation

Step 7 of Figure 2 is the application of a criterion – in this case Interface Mutation – to select test data from the input domain. The selection of test cases using Interface Mutation was limited to the subdomains created.

The selection of test cases from the subdomain is described in Figure 3. First a small set of mutants is generated using a few Interface Mutation operators. The test cases in the subdomain – numbered from 1 to 2000 – are picked in sequence and used to execute these mutants. Effective test cases – those that distinguish at least one live mutant – are retained in the test set, the others are discarded. In this way we select only those test cases that would improve the test set according to the Interface Mutation criterion. This step is repeated until a selected test case reveals a fault or until the entire subdomain has been exhausted without revealing a fault. Some fault-revealing test cases may not be selected because they do not distinguish any live mutants.

If the entire subdomain is exhausted and none of the selected test cases is able to reveal a fault then new mutants are generated and the process is repeated. In this way an incremental strategy for applying Interface Mutation is determined. If all the mutants, according to such strategy, are generated and a fault is not revealed then the test ends without success.

The sequence of mutation operators applied in this experiment is the following:

- Group II operators
- Interface variables replacement operators (type "DirVarRep")
- Interface variables increment and decrement operator (type "DirVarIncDec")

- Unary operators insertion at interface variable references operators (type "DirVarxxxNeg)
- *return* statements operators (type "RetSta")
- Non-interface variables replacement operators (type "IndVarRep")
- Non-interface variables increment and decrement operator (type "IndVarIncDec")
- Unary operators insertion at non-interface variable references operators (type "IndVarxxxNeg")

Using the parameterization characteristics avaiable in PROTEUM/IM, an approach called "upper-bound" was adopted to reduce the number of mutants to be analyzed. The first 5 groups of operators listed above had the number of mutants per application point restricted to at most 2 mutants. The other groups had that number restricted to at most 1 mutant. In addition, only 10% of the mutants were randomly selected. Thus the two approaches were combined, limiting the number of mutants at each application point and then selecting only 10% of those mutants. With these two approaches a reduction of approximately 94% mutants is achieved. For the $SPACE_{\Phi}$ version the number of mutants generated for the same connections using the operators described would be 91352. With the upper bound approach and 10% random selection this number reduces to 5129.

In addition, some connections were avoided during mutant generation. For the following connections no mutants were generated: 1) main-glvainit; 2) glvainit-kwdsinit; 3) glvainit-emsginit; 4) glvainit-unaminit; and 5) glvainit-moseinit. These connections are executed only once in each program run. The called functions are responsible only for the initialization of some static tables. Hence we decided not to apply Interface Mutation to these connections. We expect to reveal faults in table initialization by testing other connections related to functions where these tables are used. The number of mutants avoided using this approach is 77789. The total number of mutants, for all operators and all connections would be 169141. Avoiding these connections this number falls to 91352.

3.6 Fault Removal

Once a test case reveals a fault, it needs to be removed from the version $SPACE_N$ in order to restart the test. This operation is represented in Step 8 of Figure 2. It should be noticed that in some cases, when a fault revealing test case is selected, more than one fault has to be removed at the same time in order to make the program behave correctly with that test case. There is also the possibility that when faults are related to each other more than one fault is a candidate for removal resulting in a version that behaves correctly with the fault revealing test case. Assumning that test case t makes version $SPACE_N$ fail, the process of fault removal is described as follows:

1. Initialy, one fault is removed each time. First, the one with the highest HI, then the next, until the removal of a fault fixes the program for the fault revealing test case or until all faults were tried unsuccessfully.
2. If no single fault could be successfully removed, then all the faults for which t leads to a failure are simultaneously removed.
3. If these efforts to remove faults fail, then faults must be removed by hand. None of our experiments required human intervention.

Finally, after executing experiments I to X, data are collected and analyzed. This data and its analyses are reported in the next section.

4 Results and Data Analysis

First we present the number of mutants generated by each of the mutation operators. It is important to note that in one experiment the number of mutants generated by the same operator may change from one version of *SPACE* to another, during its "evolution". Table 6 shows the number of mutants created for the $SPACE_{\Phi}$ version.

Table 6: Number of mutants per operator for the $SPACE_{\Phi}$ program

Operator	Mutant	Operator	Mutant
I-DirVarAriNeg	93	I-DirVarBitNeg	70
I-DirVarIncDec	700	I-DirVarLogNeg	77
I-DirVarRepCon	217	I-DirVarRepExt	101
I-DirVarRepGlo	4	I-DirVarRepLoc	341
I-DirVarRepPar	240	I-DirVarRepReq	165
I-IndVarAriNeg	248	I-IndVarBitNeg	229
I-IndVarIncDec	484	I-IndVarLogNeg	222
I-IndVarRepCon	278	I-IndVarRepLoc	293
I-IndVarRepPar	484	I-IndVarRepReq	250
I-RetStaDel	91	I-RetStaRep	95
II-ArgAriNeg	22	II-ArgBitNeg	13
II-ArgIncDec	270	II-ArgLogNeg	22
II-ArgRepReq	28	II-ArgStcAli	20
II-FunCalDel	72	TOTAL	5129

Table 7 presents, in increasing order of HI, the faults selected for Experiment I. As HI depend on the subdomain for which it is calculated, each experiment may have different sets of faults and different HI for the same fault.

Table 7: Faults inserted in *SPACE* for Experiment I

FAULTS	33	12	18	22	27	8	17	7	
HI	0.05	0.15	0.15	0.25	0.4	1.4	1.65	1.75	
FAULTS	20	21	23	16	24	29	13	11	Average
HI	2.55	2.55	2.55	4.85	5.6	6.25	6.65	9.9	2.92

Table 8 shows the evolution of the test for Experiment I. Each row represents one phase of the test. The first column presents the number of faults during each phase. The second column shows the number of test cases selected up to the point when a test case reveals a fault or until the point it was not possible to reveal a fault. The third column shows the number of faults chosen to be removed at the end of that phase. The fourth column shows the number of test cases selected in the phase and the last column shows the mutation score obtained by those test cases.

For Experiment I, the test was initiated with 16 faults and 431 mutants generated in the first phase. It was sufficient to select only one test case to reveal a fault. Then, in order to fix the program and make it behave as the oracle with that test case, it was necessary to remove faults 23, 16 and 24. The mutation score obtained by that single test case was 0.1787. The second line shows the number for the new version, with only 13 faults, and so on.

Tables 10 to 27 in Appendix B show information in columns same as that in Tables 7 and 8 for experiments II to X. In the column "Faults Removed" sometimes the numbers are enclosed in parenthesis implying that those faults could not be revealed at the end of the test. For example, fault 33 in Experiment III, Table 13 could not be revealed until the end of the test.

In all 159 faults were reinserted in the *SPACE* program. The average of the difficulty to reveal those faults – expressed by their Hardness Index – range from 2.36% to 3,79%. Only 4 out of the 159 faults were not revealed. These are fault 33 in Experiments III and VI and faults 17 and 7 in

Table 8: Results of Experiment I

Number of faults	Mutants generated	Faults removed	Test cases	Mutation score
16	431	23, 16, 24	1	0.1787
13	427	13	7	0.2998
12	436	8	8	0.3050
11	436	7, 11	12	0.3899
9	437	29	13	0.4005
8	437	27	15	0.4462
7	437	20, 21	27	0.5515
5	437	12, 18	43	0.6293
3	435	33, 22	50	0.6759
1	2215	17	61	0.4790
0	5129	–	85	0.3931

Experiment VII. The chances of selecting fault revealing test cases for these 4 faults are just 0.05%, 0.05%, 1.45% and 1.55%. In addition, none of the faults passed unrevealed in all the 10 repetitions of the experiment. Thus, as a main conclusion we can say that the application of Interface Mutation tends to select fault revealing test cases when they exist.

Analyzing each experiment separately we can see that in 3 of them it was not possible to reveal all the inserted faults. In experiments III and VI, fault number 33 was not revealed. In both experiments, only one test case in the subdomain would reveal the fault but they were not selected by the Interface Mutation. However, in other experiments like I, VII and X, the same fault had the same Hardness Index of 0.05% and was revealed. This fact is related with the way test cases are selected from the subdomain. Each test case has a number from 1 to 2000 and is used according to this order. Thus, if a fault revealing test case is at the beginning of the subdomain it has a larger probability of being selected because the number of live mutants that would be executed with this test case is also larger. If the test case falls towards the end of the subdomain, its chance of being selected is smaller because most of the mutants that it could distinguish would most likely be already distinguished by some other test case. In experiments III and V, the only test cases that would reveal fault 33 were at positions 1470 and 1276 of their respective subdomains.

In experiment VII faults 17 and 7 were not revealed but were revealed in all other experiments. However, in Experiment VII, the number of revealing test cases for either faults is much larger: 29 for fault 17 and 31 for fault 7. This means that when test case 54 was tried – the first one that reveals fault 17 – all the mutants that this test case would distinguish had already been distinguished by test cases 1 to 53. The same happens for the other fault revealing test cases. Table 9 shows the sequence of selected test cases for experiment VII and the fault 7 and fault 17 revealing test cases. Only one test case reveals both faults, leading to a set of 59 test cases that could reveal either fault. Surprisingly, none of those 59 test cases were selected by the application of Interface Mutation.

The observation that the order of selection of test cases can influence the criterion effectiveness suggests that we could use some approach to try to raise the probability to select fault revealing test cases. Each mutant determines a subdomain in the input domain, composed by the test cases that distinguish that mutant. A fault revealing test case in this subdomain has a probability of being selected. If it is alone in that subdomain, this probability is 1 and this is a fault revealing mutant. If there are other test cases in the subdomain, the chance of being selected is shared with the other – possibly not fault revealing – test cases. A way of trying to increase the chances to select a fault revealing test case is to establish a "strength factor" for the mutants. This factor determines that, in order to kill this mutant it should be distinguished by n test cases, $n \geq 1$. It means that more than one test case could be selected in each subdomain, depending on the strength factor. This approach increases the number of mutant executions required in a program test but might not elevate too much the number of required test cases. Consider two mutants A and B and two test cases t_1 that

Table 9: Selected test cases and fault revealing test cases for faults 17 and 7 in experiment VII

Selected test cases	Fault 7 revealing test cases	Fault 17 revealing test cases
1 2 4 5 6 7 8 9 10 11 12	226 339 397	54 82 340
14 15 17 20 21 22 29 30 33	458 474 512	453 464 581
40 44 45 50 55 56 58 65 68	759 760 790	616 671 759
81 85 86 97 98 100 107 108	828 915 968	864 903 907
123 126 133 134 165 177 179	1041 1132 1207	927 938 961
184 215 218 224 230 277 299	1283 1316 1332	1060 1098 1191
319 329 359 435 477 489 493	1341 1342 1380	1213 1324 1494
524 533 614 660 808 839	1522 1658 1663	1527 1545 1584
1218 1540 52 113 274 365	1715 1718 1736	1589 1613 1810
609 881 1369 1724 187 311	1869 1877 1880	1819 1854
1329 1889 1617 697 3 64 78	1890	
120 349 536 653 1560 1949		
110 1428 144 939		

distinguishes A and t_2 that distinguishes B. This means that A has been executed with t_1 but not with t_2 and B has been executed with both t_1 and t_2. Using a strength factor of 2, A is executed with t_2 and it is possible that t_2 distinguishes A, so we have got 2 distinguishing test cases required for A. Then it is necessary to find one more test case to distinguish B. So, in the best case, only one new test case is required for any set of mutants. In the worst case – that is as unlike as the best case – one extra test case for each mutant is required.

Another idea is to establish a "stress factor" for the test case set. After distinguishing n mutants, a test case looses its power and is no longer used, requiring new test cases to distinguish the remaining mutants. In this case, the cost to execute the mutants does not change but we can expect a larger test case set. This approach atempts to avoid that a single test case distinguish a large number of mutants preventing other – possibly fault revealing – test case from being selected. Thus, a chance is given to other test cases. The strength factor as well as the stress factor may be useful mainly if random generation is used to produce test cases to distinguish the mutants.

In terms of cost, this study shows that with a reduced number of mutants – around 5100 mutants for a program with 4500 LOC – reasonably good effectiveness results were achieved. In addition, most faults could be revealed with a even smaller set of mutants. From the 159 faults inserted in the 10 repetitions of the *SPACE* experiment, 89.31% (142 faults) were revealed by the initial mutant set, generated only by the application of operators from Group II. This initial set has at most 447 mutants. In the next step, using a set of 1515 mutants, 5 more faults were revealed. With 2215 mutants, 8 more faults were revealed and with 3946 mutants, 1 more fault was revealed. Above 3946 up to 5129 mutants, that is the maximum number of mutants used, no fault was revealed and four faults remained unrevealed. This data shows the efficacy of the incremental approach used in the application of Interface Mutation. Most faults could be revealed with a highly very reduced mutant set, i.e. with a reduced cost. In addition, by improving the mutant set it was possible to reveal most of the hard-to-reveal faults.

5 Conclusion

Interface Mutation is a new approach that uses mutation technique to evaluate test sets at interprocedural level. Previous work [4] has indicated that Interface Mutation is potentialy a good alternative for application at integration level. It has been shown to be fault-revealing and cost effective. In this paper we developed a more complete case study aiming at evaluating the cost and effectiveness of Interface Mutation by applying it in a more complex (*SPACE*) program and atempting to reproduce a real testing environment.

We selected the *SPACE* program because it is a used, non trivial program developed for the European Space Agency (ESA). In addition, it has a development record that kept track of all the faults uncovered during its development cycle. Using this information we reproduced in the experiment a testing process, starting with a version of the *SPACE* with several faults and then applying Interface Mutation to select test case and try to reveal the faults. This way we could evaluate the cost and the fault revealing effectiveness of the Interface Mutation criterion.

The results show that Interface Mutation is an effective, reasonable-cost criterion. With a relative reduced number of mutants (5129), only 4 out of 159 faults re-inserted in *SPACE* could not be revelead by the application of Interface Mutation. In addition, the use of an incremental application of Interface Mutation operators has shown that with even smaller sets of mutants, many faults can be revealed. For example, using a highly restricted set of Interface Mutation operators that generate nearly 450 mutants, around 90% of the faults could be revealed.

Based on our experience, we believe that Interface Mutation constitutes a valuable instrument for integration testing. We intend to explore the concepts of Interface Mutation in other testing domains such as:

- object-oriented class integration;
- interface specification using Interface Definition languages; and
- in client/server applications.

Currently we are evaluating the Interface Mutation operators to determine an essential set. We are also formulating an incremental, cost-effective integration testing strategy by investigating the complementary aspects of mutation testing during unit and integration testing phases. Other related issues we want to explore in the future are the study of the influence of software architecture and integration strategy to the applicability of Interface Mutation and its relation to other interprocedural criteria.

References

[1] Delamaro, M. E.: 1997, "'Mutação de Interface: Um Critério de Adequação Inter-procedimental para o Teste de Integração"'. Doctoral dissertation, Physics Institute of São Carlos - University of São Paulo, São Carlos, SP.

[2] Delamaro, M. E. and J. C. Maldonado: 1996, "'Proteum - A Tool for the Assesment of Test Adequacy for C Programs"'. In: *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*. Brunswick, NJ, pp. 79–95.

[3] Delamaro, M. E., J. C. Maldonado, and A. P. Mathur: 1996, "'Integration Testing Using Interface Mutation"'. In: *Proceedings of the VII International Symposium of Software Reliability Engineering (ISSRE'96)*. White Plains, NY, pp. 112–121.

[4] Delamaro, M. E., J. C. Maldonado, and A. P. Mathur: 1998, "'Interface Mutation: An Approach for Integration Testing"'. Submited to the IEEE Trans. on Software Engineering.

[5] DeMillo, R. A., R. J. Lipton, and F. G. Sayward: 1978, "'Hints on Test Data Selection: Help for the Practicing Programmer"'. *IEEE Computer* **11**(4).

[6] DeMillo, R. A. and A. J. Offutt: 1991, "'Constraint Based Automatic Test Data Generation"'. *IEEE Transactions on Software Engineering* **17**(9), 900–910.

[7] Frankl, P. G. and E. J. Weyuker: 1993, "'A Formal Analysis of the Fault-Detecting Ability of Testing Methods"'. *IEEE Transactions on Software Engineering* **19**(3), 202–213.

[8] Harrold, M. J. and M. L. Soffa: 1991, "'Selecting and Using Data for Integration Test"'. *IEEE Software* **8**(2), 58–65.

[9] Mathur, A. P. and W. E. Wong: 1994, "'An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria"'. *The Journal of Software Testing, Verification, and Reliability* **4**(1), 9–31.

[10] Offutt, A. J.: 1989, "'Coupling Effect: Fact or Fiction"'. In: *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification.* Key West, FL, pp. 131–140.

[11] Offutt, A. J., G. Rothermel, and C. Zapf: 1993, "'An Experimental Evaluation of Selective Mutation"'. In: *Proceedings of the 15th International Conference on Software Engineering.* Baltimore, MD, pp. 100–107.

[12] Pasquini, A.: 1996, "'SPACE Program"'. Personal correspondence.

[13] Rapps, S. and E. J. Weyuker: 1985, "'Selecting Software Test Data Using Data Flow Information"'. *IEEE Transactions on Software Engineering* **SE-11**(4), 367–375.

[14] Wong, W. E.: 1993, "'On Mutation and Data Flow"'. PhD dissertation, Department of Computer Science, Purdue University, W. Lafayette, IN.

[15] Wong, W. E., J. R. Horgan, S. London, and A. P. Mathur: 1994a, "'Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness"'. In: *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering.* Monterey, CA, pp. 230–238.

[16] Wong, W. E. and A. P. Mathur: 1995a, "'Fault Detection Effectiveness of Mutation and Data Flow Testing"'. *Software Quality Journal* **4**(1), 69–83.

[17] Wong, W. E. and A. P. Mathur: 1995b, "'Reducing the Cost of Mutation Testing: An Empirical Study"'. *The Journal of Systems and Software* **31**(3), 185–196.

[18] Wong, W. E., A. P. Mathur, and J. C. Maldonado: 1994b, "'Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength, and Effectiveness"'. In: *Proceedings of the International Conference on Software Quality and Productivity.* Hong Kong, pp. 258–265.

A Features of the *SPACE* Program Execution

At each execution the program asks for the name of the file that contains the description to be processed and generate another file with the list of elements positions and excitations. An example of the input file is shown in Figure 4 and an example of the output file is shown in Figure 5.

```
GROUP testgroup

    GRID RECTANGULAR

    ELEMENT
        GEOMETRY RECTANGULAR 20,10 cm

    ADD BLOCK -1,-1 1,1
    ADD NODE -2,-2
    ADD NODE -2,2
    ADD NODE 2,2
    ADD NODE 2,-2

    GROUP_EXCITATION
        AMPLITUDE UNIFORM
        PHASE UNIFORM

END
```

Figure 4: Input file for the *SPACE* program

```
**************************************************************************
            ARRAY PREPROCESSOR ver.  1.0
**************************************************************************

Enter source file name (omit .ADL) ?:

    2     GROUP testgroup
    4     GRID RECTANGULAR
    6     ELEMENT
    7     GEOMETRY RECTANGULAR 20 10 cm
    9     ADD BLOCK -1 -1 1 1
    10     ADD NODE -2 -2
    11     ADD NODE -2 2
    12     ADD NODE 2 2
    13     ADD NODE 2 -2
    15     GROUP_EXCITATION
    16     AMPLITUDE UNIFORM
    17     PHASE UNIFORM
    19     END

* Generating GROUP: testgroup.
* PSTEP,QSTEP set to:200.000000,100.000000
* Grid ANGLE set to:90.000000
* Group ROT_ANGLE set to:  0.000000

* Writing output file:  gr1.dat

    NEL
 NPORTS
   IDUM      REXA      REYA   REZA     THEA   PHEA   PSEA
              AMP       PSH    PSC   PHEPOL
     13
      1
      1   -200.00   -100.00   0.00     0.00   0.00   0.00
            1.00      0.00   0.00    90.00
      2     0.00   -100.00   0.00     0.00   0.00   0.00
            1.00      0.00   0.00    90.00
             ...       ...    ...      ...    ...    ...
     12   400.00    200.00   0.00     0.00   0.00   0.00
            1.00      0.00   0.00    90.00
     13   400.00   -200.00   0.00     0.00   0.00   0.00
            1.00      0.00   0.00    90.00
* End writing output file:  gr1.dat
```

Figure 5: Output file for the *SPACE* program

B The Complete Experiment Data

Table 10: Faults inserted in *SPACE*for Experiment II

FAULTS	12	18	22	27	8	7	17	20	21
HI	0.1	0.1	0.25	0.25	0.8	1.8	2.0	2.55	2.55
FAULTS	23	13	16	29	24	11	31	10	Average
HI	3.00	5.5	5.5	5.5	6.45	8.65	9.55	9.8	3.79

Table 11: Results of Experiment II

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
17	439	8, 7, 23	4	0.2187
14	439	11	11	0.3303
13	431	22, 24	15	0.4640
11	433	29	17	0.4111
10	433	10	18	0.4273
9	441	17	33	0.5374
8	441	20, 21, 16	37	0.5488
5	1491	13, 31	49	0.4071
3	437	12, 18	44	0.6347
1	2214	27	64	0.4770
0	5129	–	87	0.3881

Table 12: Faults inserted in *SPACE* for Experiment III

FAULTS	33	27	12	18	22	8	20	21	
HI	0.05	0.1	0.15	0.15	0.2	1.25	1.9	1.9	
FAULTS	7	17	23	16	29	13	24	11	Average
HI	2.1	2.15	3.6	5.05	6.2	6.6	6.85	9.55	2.99

Table 13: Results of Experiment III

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
16	431	13	1	0.1740
15	433	29	4	0.2079
14	433	11	6	0.2702
13	432	24	16	0.3819
12	437	20, 21, 16	19	0.4691
9	437	8, 23	33	0.5378
7	437	7	36	0.5744
6	437	17	39	0.5995
5	437	12, 18	43	0.6201
3	435	22	48	0.6575
2	2214	27	69	0.4883
1	5129	(33)	96	0.3997

Table 14: Faults inserted in *SPACE* for Experiment IV

FAULTS	27	12	18	22	8	7	20	
HI	0.1	0.2	0.2	0. 2	1.25	1.35	1.85	
FAULTS	21	17	23	16	24	13	29	Average
HI	1.85	2.1	2.25	3.8	5.5	6.0	6.35	2.36

Table 15: Results of Experiment IV

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
14	430	17	5	0.2372
13	430	29	9	0.2814
12	430	7, 23, 16, 24	11	0.3907
8	437	13	23	0.5240
7	437	20, 21	32	0.5835
5	437	22	40	0.6316
4	437	12, 18	48	0.6270
2	1515	8	66	0.4607
1	2214	27	72	0.4865
0	5129	–	96	0.3964

Table 16: Faults inserted in *SPACE* for Experiment V

FAULTS	27	12	18	22	8	26	7	17	21
HI	0.15	0.25	0.25	0.5	1.1	1.65	1.65	1.75	2.15
FAULTS	20	23	10	16	29	13	24	11	Average
HI	2.2	2.65	3.95	4.65	5.3	6.05	6.65	9.0	2.94

Table 17: Results of Experiment V

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
17	439	7, 17, 23	1	0.1617
14	439	22, 11	6	0.4146
12	439	29	7	0.3394
11	439	24	8	0.3440
10	433	10	9	0.3626
9	441	26	15	0.4603
8	441	13	17	0.4535
7	437	27	17	0.4554
6	437	16	15	0.4371
5	437	12, 18	44	0.6156
3	447	21, 20	46	0.6264
1	447	8	55	0.6577
0	5129	–	98	0.4042

Table 18: Faults inserted in *SPACE* for Experiment VI

FAULTS	27	33	12	18	22	8	20	21	7
HI	0.05	0.05	0.1	0.1	0.6	1.15	1.3	1.35	1.45
FAULTS	17	23	16	29	13	24	11	10	Average
HI	1.65	2.85	4.4	6.15	6.95	7.15	8.75	9.9	3.17

Table 19: Results of Experiment VI

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
17	439	13	2	0.2255
16	434	24	4	0.2512
15	442	17	9	0.2986
14	442	11	10	0.3213
13	436	7, 23	10	0.3395
11	436	10	13	0.3784
10	437	29	15	0.4005
9	437	8	26	0.4508
8	435	20, 21, 16	31	0.5149
5	437	22	39	0.6270
4	437	12, 18	44	0.6087
2	2214	27	75	0.4846
1	5129	(33)	97	0.3944

Table 20: Faults inserted in *SPACE* for Experiment VII

FAULTS	33	12	18	27	22	8	17	7	
HI	0.05	0.3	0.3	0.3	0.5	0.7	1.45	1.55	
FAULTS	20	21	23	16	13	29	24	11	Average
HI	1.65	1.65	2.95	5.0	5.95	6.25	6.9	8.5	2.75

Table 21: Results of Experiment VII

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
16	431	29	2	0.1671
15	431	24	4	0.2575
14	427	11	10	0.3489
13	437	27	16	0.3570
12	437	16	4	0.2609
11	437	13	27	0.4874
10	437	8, 23	33	0.5057
8	437	20, 21	38	0.5286
6	437	22	50	0.6362
5	437	12, 18	63	0.6568
3	1515	33	72	0.4673
2	5129	(17,7)	93	0.4022

Table 22: Faults inserted in *SPACE* for Experiment VIII

FAULTS	12	18	27	22	8	17	20	21
HI	0.15	0.15	0.15	0.5	1.35	1.75	1.75	1.75
FAULTS	7	23	16	29	13	24	11	Average
HI	1.75	3.35	4.8	6.35	7.1	7.35	9.65	3.19

Table 23: Results of Experiment VIII

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
15	431	13	1	0.1508
14	433	29	7	0.2771
13	433	7, 11	15	0.3903
11	432	24	27	0.4815
10	437	8, 23	30	0.4851
8	437	20, 21, 16	32	0.5034
5	437	12, 18	45	0.5858
3	435	22	50	0.6460
2	2214	27	67	0.4765
1	3946	17	80	0.4098
0	5129	–	86	0.3880

Table 24: Faults inserted in *SPACE* for Experiment IX

FAULTS	12	18	27	22	8	7	17	20
HI	0.2	0.2	0.2	0.25	0.8	1.25	1.3	1.85
FAULTS	21	23	16	24	13	29	11	Average
HI	1.85	2.15	4.7	6.05	6.4	7.25	8.9	2.89

Table 25: Results of Experiment IX

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
15	431	13, 11	2	0.1833
13	432	23 16 24	3	0.2362
10	437	29	6	0.3272
9	437	22, 7	23	0.5858
7	437	20 21	27	0.5492
5	437	27	31	0.5767
4	437	8	35	0.5995
3	437	17	45	0.6293
2	437	12, 18	47	0.6362
0	5129	–	89	0.3972

Table 26: Faults inserted in *SPACE* for Experiment X

FAULTS	33	12	18	22	27	8	7	17	
HI	0.05	0.25	0.25	0.25	0.3	0.65	1.05	1.5	
FAULTS	20	21	23	16	24	29	13	11	Average
HI	1.9	1.9	2.45	4.4	6.35	6.65	6.65	9.55	2.76

Table 27: Results of Experiment X

Number of faults	Mutants generated	faults removed	Test cases	Mutation score
16	431	11	6	0.2784
15	430	20, 21, 16	7	0.3023
12	430	7, 23, 29	10	0.3581
9	430	22, 24	13	0.4884
7	441	13	15	0.4830
6	437	12, 18	23	0.5378
4	447	8	25	0.5526
3	447	17	42	0.6175
2	1515	33	55	0.4548
1	2214	27	63	0.4788
0	5129	–	81	0.3890