ISSN 1744-1986

The Open University

Technical Report N^{o} 2007/09

The Csaw Mutation Tool Users Manual

Michael Ellims

3rd July 2007

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

The Csaw Mutation Tool Users Manual

MICHAEL ELLIMS

mike.ellims@pi-shurlok.com

Mutation is a technique that holds great promise in testing research, however it can be difficult getting hold of tools to allow the application of the technique to programs written in the C programming language. One tool that is available tool is Proteum. Here we describe another small tool set that can be used for performing mutation on C functions and provide guidance for the using the tool.

Categories and Subject Descriptors: D.2.5 Testing and Debugging
General Terms: Testing tools
Additional Key Words and Phrases: C programming, mutation

Introduction

This document provides basic information on using the Csaw mutation tool set. It should be noted from the outset that the Csaw tool set is a research tool and as such has not been developed with the same process rigor that a commercial tool *may* have been. However the price is right compared with commercial tools and it is possibly worth at least what you'll pay for it.

In addition to providing information on how to use the tool, this manual also provides minimal design information and background to some of the design decisions. However the majority of the information on the construction of the tool will have to be gleaned from the code files that make up the system.

The tool itself comprises two main parts, one to mutate code and one to take the mutated code and execute it which in itself is not a trivial problem. Mutations can result in code that will crash because of a divide by zero and similar problems, or with execute for an infinite length of time (common with code that contains loops). The driver program to execute the mutated code has to take this into account.

Overview

The tool itself has some limitations that may make it unattractive to the hardcore user. Firstly it was primarily designed to operate on a single function. This decision was made because the reason for developing the tool was to be able to compare different methods for automatically generating input test data for unit (i.e. C function) testing. There is some limited support for dealing with called functions, but this has not been extensively developed or tested. The second major limitation is that the tool process the input test one line at a time in a similar manner to the tool used by Andrews et al. [2] and as such does not parse the code other than to recognise symbol and keywords and build up symbol tables of variable names and constants. However this does have one advantage, the tool can potentially be used with C++ or Java programs or any code which has a structure similar to the C language[1].

A side effect of these two limitations was that that several special tags have been introduced to allow additional information to be supplied to the tool such as the name of the functions being mutated and information on global or static variables with file scope that are not defined in the function being mutated.

There are other limitations that can annoying, for example the tool does not deal with comments in the code at all. Therefore code that is presented to the tool must be stripped of all comments or the text that they contain often ends up being defined in the symbol table as constant data. Another annoyance is that because the code is not parsed, names for structure elements are treated as unique variable identifiers which can result in a large number of mutants that cannot be compiled as all name tokens are substituted by brute force i.e. all scalar names are substituted for each other and all array names are substituted for each other.

Other limitations concern the types of data that the tool can be used on. The primary target for the tool is integer code as used in real-time embedded systems so support for floating point arithmetic is currently basic and for string processing nonexistent. However this type of code (i.e. integer based) forms a large proportion of the code used in testing research and is probably not an insurmountable barrier to the tools future application.

Having mentioned some of the tools limitations we now turn to what the tool can do.

[1] Comment from J. Andrews (personal communication).

Operator mutations

The tool can swap one operator for another e.g. '+' for '`*`', and so on. It supports this for arithmetic and logical operators, variable types etc. The substitutions that can be applied are defined in tables so can be easily altered. Some operators are not substituted e.g. '.' and '`->`' are not currently defined in the tables as there mutants often result in code that cannot be compiled.

Variable substitution

The tool will swap one variable name for another variable name e.g. if the variables `i`, j and `k` are defined then all instances of `i` will be swapped with `j` and `k`, all instances of `j` with `i` and `k` and so on. Variable substitution is done on scalar and array types independently.

Constant substitution

All textual constants (e.g. from #define) are swapped for all other such defined constants that the tool finds in the text of the function being mutated. The tool considers a constant any text string that is not recognised as either a keyword or a variable.

Decimal constants

The tool will offset decimal constant values by plus or minus one, for example a constant of "10" will be converted to "both "9" and "11". Floating point, double precision and hexadecimal constant mutations have not currently been implemented. However "holes" have been left in the code for them.

Array Index Mutation

Array indexes that use variables are mutated by appending either "+1" or "-1", so `array[i]` is converted to `array[i-1]` and `array[i+1]` to produce off by one errors. Not variable substitution also affects array indexing.

Statement removal

Each statement that ends in a '`;`' is deleted, however for this to work correctly a statement must be on a single line.

Type mutations

Unlike any other system the author knows of, this tool will mutate the type specifier of a variable, so it will swap "`unsigned int`" for "`int`" , or "`double`" for "`float`" etc. This capability has been introduced because the primary target for the tool is integer based, real-time embedded code. Because of the limited amount of memory that this type of system often has, it is common to use integers of the smallest possible size e.g. using a `char` or `unsigned char` for a integer variable which only takes on a small number of values. Like operator mutations, type mutations are defined in an extendable table.

Related work

This section cites the published work that has had the most influence on the creation of the Csaw tool. Note that the author has become acquainted with this area from the study of automatic test data generation via the Godzilla tool automatic test generation tool and its association with Mothra mutation system (references below). This has resulted in a what possibly slightly eclectic set of references.

Code mutation (and other forms of mutation) have a long history. The first work in this area was from Hamlet [15]. However the work usually cited is by De Millo et al. [11] where the concept of introducing mistakes that approximate what a competent programmer may make, that is "programs that are nearly correct".

The early work with mutation tools has been with the FORTRAN language performed by King and Offutt [23], [16] with the Mothra testing tool. Closely allied to this work was the Godzilla tool chain used to build constraint systems, using these to automatically create test sets DeMillo and Offutt [26], [12],[13]

Offut has been active in continuing work on mutation which has delivered a number of useful results. Much of this work is been directed at reducing the workload that is associated with the technique.

Using the concept of selective mutation proposed by Mather [19] Offutt et al. [22], [24] examined the question of whether all of the mutation operators were required and reached the conclusion that possibly they were not. Of the 22 operators in Mothra, five (ABS, AOR, LCR, ROR and UOI) seemed sufficient to enable the Godzilla tool set to generate tests that would kill the majority of all possible mutants. The importance of these results is that it suggests that the fact that Csaw does not produce all possible mutations is perhaps not that serious. However no detailed investigation into this has been undertaken to data.

Offutt [21] extended the concept of selective mutation is combined with the proposal that it may valid to ignore the small number of mutants not killed (assuming a Godzilla like tool for generating tests) to propose a more efficient way of generating final test sets. The system structure suggested in this work is a major driver in the development of this tool. However in ongoing research a test data generation mechanism based on the AETG algorithm [6] replaces the Godzilla tool.

One further results of importance comes from an investigation by Offutt & Voas [25] on the relationship between mutation adequacy and coverage metrics, showing that mutation subsumes most commonly used metrics. Thus mutation adequate tests should also be at least branch adequate.

More recently Andrews et al. [2] conducted experiments to determine if mutation faults were a realistic approximation to real faults and reached an affirmative conclusion, lending support to the competent programmer hypothesis. An major result of this work was that hand seeded faults appear more difficult to detect than both mutation induced faults and importantly faults actually found in real code. This work is extended in [3] to investigate the relationships between different coverage criteria.

Agrawal et al. [1] produced a technical report on mutant operators for the C language. A summary of those operators and how they compare with operators at Csaw uses is given in Table 1. The first column gives the type of object operated on, the second column (operator) the mutation name used in [1] and the third column is a short of description of the operators actions. The final column is an assessment of how closely the Csaw tools come to implementing the operator.

Table 1: Summary of mutation operators and comparison with Csaw mutation tool. Notes on equivalent Mothra mutation operates are included in usage column.

Area	Operator	Usage	Csaw
Statement	STRP	trap on statement execution, replaces each statement with code to cause a termination	no
	STRI	trap on if condition, replaces branch predicate with code to cause termination	no
	SSDL	statement deletion	implemented
	SRSR	`return` statement replacement	no
	SGLR	`goto` label replacement	no
	SCRB	`continue` replacement by `break`	no
	SBRC	`break` replacement by `continue`	no
	SBRn	`break` to nth enclosing level	no
	SCRn	`continue` to nth enclosing level	no
	SWDD	while replaced by do-while	no
	SDWD	do-while replaced by while	no
	SMTT	multiple trip trap, used to ensure that a loop is executed more than once.	no
	SMTC	multiple trip continue, ensure that if a loop executes n iteration on the n+1 it will not execute the body.	no
	SSOM	sequence operator mutation, used to modify effect of the comma operator	no
	SMVB	move closing brace up or down one line	no
	SSWM	switch statement mutation, cause execution to halt if a case is selected	no
Operators	Obom	binary operator mutation (OAAN, Mothra AOR) (OBBN, Mothra LCR) (ORRN, Mothra ROR)	implemented
	OUOR	unary operator mutation	implemented
	OLNG (UOI)	logical negation	indirectly
	OCNG	logical context negation	partly
	OBNG	bitwise negation	indirectly
	OIPM	indirect operator precedence mutation	no
	OCOR	cast operator replacement	indirectly

Variables	Varr	mutate array references in expressions	implemented
	Vprr	mutate pointer references in expressions	partial
	Vsrr	scalar variable reference replacement	implemented
	Vtrr	mutate structure references	partial
	VASM	mutate subscripts in array references (multi dimensional arrays)	no
	VSCR	mutate components of structure	indirectly
	VDTR	variable domain traps, TRAP on negative, zero and positive (Mothra ABS)	no
	VTWD	twiddle mutations, mirror off by one errors e.g. +/- 1 (Mothra UOI)	partial
Constants	CRCR	required constant replacement	no
	CCCR	constant for constant replacement	partial
	CCSR	constant for scalar replacement	no

The utility of some operators are lower than for others, for example the SRSR operator is most effective if there are multiple return statements as variable replacement operators will generally modify a statement of the form `return (xyz)`. Likewise the `goto` replacement label operator SGLR is only useful if the `goto` is actually used.

Some operators are impossible for the Csaw tool set to implement, for example SMVB operator normally operates over multiple lines, so Csaw, which operates one line at a time cannot deal with this.

As shown in Table 1 some rules are “indirectly” implanted. This means that while the exact mutation mechanism as suggested in [1] is not used, Csaw achieves a similar effect via brute force though it should be noted Csaw’s version may not have exactly the same properties. For example the Varr operator is type aware i.e. it does not substitute integer arrays for pointer arrays but Csaw will happily, the compiler will catch many instances such as this so the net effect will be almost the same. Another example is structure component replacement, Csaw achieves nearly the same end result by using every possible variable it knows about. The vast majority will not compile but this still effectively sifts them out.

Agrawal et al. [1] note that a tool Proteum has been produced by a group headed by Professor Jos`e Maldonado at the University of Sao Paulo. The author first attempted to obtain this tool in 2002 but was unable to find the contact details. Agrawal et al. [1] provide contact details but I have not to date attempted to obtain the tool.

A small body of work has been performed using the Proteum tool and references are listed below.

Delamaro and Maldonado [7] PROTEUM: A Tool for the Assessment of Test Adequacy for C Programs

Wong et al. [29] Use of Proteum to Accelerate Mutation Testing in C Programs.

Delamaro et al. [10] Proteum/IM 2.0: An Integrated Mutation Testing Environment.

Maldonado et al. [18] Proteum: a Family of Tools to Support Specification and Program Testing Based on Mutation.

Barbosa, Maldonado and Vincenzi, [4] Toward the Determination of Sufficient Mutant Operators for C. Software Testing

Delamaro et al.[8] Interface Mutation: An Approach for Integration Testing.

Delamaro et al. [9] Interface Mutation Test Adequacy Criterion: An Empirical Evaluation.

VincenzI at al. [27] Unit and Integration Testing Strategies for C Programs Using Mutation-Based Criteria.

Vincenzi et al. [28] Bayesian-learning based guidelines to determine equivalents mutants.

Maldonado and Barbosa [17] Establishing a Mutation Testing Educational Module based on IMA-CID I.

Namin and Andrews [20] Finding Sufficient Mutation Operators via Variable Reduction.

Design Notes

line.c

The mutation tool set was designed based on the principle expounded by Ken Thompson that, “When in doubt, use brute force” [5]. There is very little that is subtle about the code.

The program `line.c` performs all of that actual mutation of code and is divided into several passes as follows.

Pass 1 : reads in the text of function(s) to be mutated along with any tag information from the file `test.c` into a 255 by 255 array of characters. It also reads a file named `nogen.txt` which blocks generation specific named (or numbered) mutants into an identical array.

Pass 2 : searches though the array containing the text of the function to mutated looking for special tags and if it finds them marks the line as “special” which excludes it from being processed by the actual mutation process in phase six. It also records the name of the target function (i.e. function to be called) during this pass.

Pass 3 : scans the code and puts any identifiers it finds into the symbol table along with any names defined by special tags. Identifiers are recognised by looking for keywords such as `int`, `float`, `static` etc. and matching the series of tokens against token patterns defined in the `decl_patterns` array defined in the `symbols.h` file. For each identifier thus identified, the line where it was defined is recorded and this information is used in the next pass.

Note that if a declaration does not have a pattern defined it won't be recognised. Therefore it is recommended that variable declarations be kept as simple as possible, for example by having one per line.

Pass 4 : refines the symbol table and assigns ranges of lines for which certain identifiers are valid. It makes use of some simplifying assumptions here, for example that a function runs from the first instance of its name to the line before the next function name. This assumption is the root cause of the requirement to define function names in tags in the order in which they appear in the text and to define all called functions before they are called.

If a variable is defined in a function declaration or inside a function then the range of lines for which it is valid falls between the two ends of the function. Thus if a function runs from lines 9 to line 27 and an identifier was located on line 12 then the valid range for that identifier is lines 9 – 27. An unfortunate side effect of this is however that duplicate names cannot be dealt with, for example if the variable “`temp`” occurred in two functions the process above would only allow it to be used in a mutation in one of the functions. To get around this change one of the names e.g. use “`temp1`” and “`temp2`”.

Pass 5 : is used to build up a symbol table of constants. The tool assumes that everything that looks like a variable name but is not in the variable symbol table is a constant. This is one of the reasons that comments have to be removed from the target code as text within the comment usually looks like a variable name. Dealing with comments is difficult as the tool only works with a single line at a time and multi-line comments would confuse it. There of course no reason that we couldn't preprocess the code read in to remove comments however this has not yet been done as in practice it is not that much of an issue. However it is recommended that the symbol tables are checked after a mutation run.

Pass 6 : performs that actual mutation of the code. To do this the code is run though one line at a time. If a line is tagged as being “special” then it is ignored. Otherwise the line is parsed to build a list of its tokens and each possible mutation of that line is written to a temporary output file named `tmp_mutant.txt`. Mutations are written to the output file `mutant.c` as follows.

```
For each line mutant in the temporary file
    Write a header comment.
    For all lines before the mutated line that are not special write them to mutant.c
    Write the current copy of the mutated line to mutant.c
    For all lines after the mutated that that are not special, written them to mutant.c
```

This results in a single file that includes all mutants for all line. This file can be very large.

In the writing process, the devil is in the detail, for example it is not possible to simply write any line directly to the `mutant.c` file as a line may contain the name of a function included in the `test.c` file. To deal with this the write process scans each line to be written and mutates any function names found so that `f_name` becomes `f_namexxxx` where `xxxx` is the four digit number assigned to each mutant. This is done for all function names allocated using the $$F and $$T tags. See the `heapSort` example to see this in practice.

driver.c

This program is used to execute the mutants under test. It includes the files `mutant.h`, `pointers.h`, created by `line.c` and the hand generated `oracle.c` program and the file containing the test vectors, `test_vectors.h`. It can also contains any files that the code under test requires to execute for example header files.

It contains two main functions, `data_select` and `test_driver`. The function `data_select` is used to sequence the tests on each mutant defined in `mutant.h` with each vector defined in `test_vectors.h` . It does this by using the system call `fork` to start a child process that calls the function `test_driver.` The function `test_driver` in turn first calls version 0 of the function under test (the unmutated code) and then calls the mutant of that function it is being compared with on the same test vectors. Finally it compares the results from both calls and decides if there is a difference, if so the mutant is "dead". The functions that actually call the target function are defined in the file `oracle.c`.

Note that the child process can fail in a number of ways, most commonly by performing a divide by zero or entering into an infinite loop. The process PID is returned from the `fork` system call and the `waitpid` system call is used to detect whether the child completed normally, crashed or had to be killed.

The `oracle.c` program is created as a separate entry to ease future automatic generation of the code contained within it.

Future work

A number of features could be altered to make the tool more usable some of a the possibilities that have been considered are listed below in no particular order.

It would be advantageous to alter the way in which the tables of declarations and operator mutations are set up. Currently this is done though static definitions of the structures used to contain the information. However as the tables get larger they also become more difficult to edit. A major upgrade will to read these tables in from text files as strings of symbols.

It is also rather annoying to have to strip of comments by hand. An additional phase could be added to the `line.c` program to deal with this in isolation but this has not yet been done.

Another possibility is to have to tool add instrumentation to the file. While gathering information on statement coverage and loop execution may be reasonable simple more detailed coverage metrics may be difficult to incorporate. One simple possibility would be to add a tag to provide a macro facility (e.g. $$M) which would allow the user to add specific instrumentation of their own devising. For example loop counters as suggested in [14].

USER MANUAL

Naming Conventions

The Csaw tool consists of several programs and a number of supporting header files that are used to manipulate the code to be mutated and to aid building a driver program for the mutations. Each of the file has a version number appended to the name so the actual file number is `name_xx.c` or `name_xx.h` etc. where "xx" is a number from 1 to 99 (e.g. 01, 02 and so on).

These numbers are *not* used in this document so the program `line_07.c` is always referenced as `line.c`, `driver_05.c` is referenced as `driver.c` and so on. In addition file names and other C names such as function names are given in `courier` font as are Linux (Unix) commands.

Programs and Header Files

The Csaw system consists of the programs and files listed below.

line.c : is the program that is used to perform the actual mutation operation on the file `test.c` that contains the text for the function `funtion_1` which is to be mutated.

mu.h : this file defines constants, enumerations and data structures that are used by the `line.c` program.

symbols.h : this file defines the constant data for keywords, symbols (e.g. "(", "=>", "<<="), special tags and the constant tables used to define operator and type specifier mutations.

test.c : Is the input file to `line.c` that contains the text of the function or functions to be mutated. Note that all the code in this file will be subject to mutation so if you want to prevent a called subroutine from being mutated it should not be placed in this file. Rather it should be included in the `driver.c` program.

nogen.txt : This file is used to suppress the generation of specific mutants by the `line.c` program. Mutants of course can be edited out of the `mutant.h` file manually but this is extremely time consuming, unbelievably mind numbing and difficult with the very large `mutant.c` programs that can be generated (tens of thousands of lines). The `split.c` program was a partial solution to the problem of editing these large files but hs been superceded by nogen.txt.

symbol_table.txt : this is the output of the symbol table that `line.c` created to determine what should be done with variables and constants. It should be checked before running the code in case `line.c` has misinterpreted any of the symbols it found or in case comments have been left in. See the `bubbleSort` example below for further details.

mutant.h : the main of the output file from the `line.c` program, it contains the text of the mutants of the code being mutated. The first "mutant" is a copy of the unmodified function that is used as the oracle in the functions defined in `oracle.c`. This file can be *extreamly* large, to ease the problem of editing this file the program `split.c` is provided to break it up into smaller sections, however the `nogen.txt` file is a faster solution.

pointers.h : this file contains a list of function pointers to the mutated versions of the target function. This file need minor editing to convert the generic function that occurs before the structure initializer to one that matches the function being mutated. The final comma in the list also need to be removed.

split.c : this program is used to break up the file `mutants.c` into smaller sections with "only" 500 mutants in each section. This was necessary as the `mutant.h` files produced by `line.c` can contain several tens of thousand lines of code. In general it has been found that text editors are not able to deal cleanly with files of this size. Now mostly redundant because of the `nogen.txt` file.

driver.c : is the driver program for the mutants under test. It includes the files `mutant.h`, `pointers.h`, `oracle.c` and `test_vectors.h` as well as any files required by the mutated code. It contains two main functions, `data_select` and `test_driver`. The function `data_select` is used to sequence the tests on each mutant defined in `mutant.h` with each vector defined in `test_vectors.h` . It does this by using the system call `fork` to start a child process that calls the function `test_driver.` The function `test_driver` in turn first calls version 0 of the function under test and then calls the mutant of that function it is being compared with on the same test vectors. Finally it compares the results from both calls and decides if there is a difference, if so the mutant is "dead". The functions that actually call the target function are defined in the file `oracle.c`.

Note that the child process can fail in a number of ways, most commonly by performing a divide by zero or entering into an infinite loop. The process PID is returned from the `fork` system call and the `waitpid` system call is used to detect whether the child completed normally, crashed or had to be killed.

oracle.c : contains two primary functions used to execute the code under test, one to call the original version of the code under test and a mutated version and another to compare the results returned. Both of these functions need to be modified to set the values that are being passed to the function, collect the results returned from both calls and to perform the compare operation.

test_vectors.h : contains the data for the test vectors to be applied to the mutants under test. It defines the test vectors as initialized arrays of type integer along with information as to the number and size of the vectors and a array of pointers to each defined test vector. The variable names `v_num_vectors`, `v_size_vector` and `v_testdata` **must** be used here.

bit_log.txt : is used to output a matrix of mutations killed vs. test vectors. Output is one vector per line, a live mutant is indicated by a '.' and a dead mutant by 'X'. An example is shown for the `bubbleSort` function.

Special Tags

There are a number of special tags that can be used in the `test.c` file before the actual text of the code being mutated is encountered, of these the "`$$T`" tag is necessary i.e. it must be included. The other tags are used to provide symbol table information that is not otherwise available to the Csaw tool.

Table 2 : tags used to provide addition information to the mutation program.

Tag	Purpose
$$V <variable name>	defines a global or static scalar variable name in the symbol table
$$A <array name>	defines a global or static array name in the symbol table
$$N <function name>	defines a global or static function name in the table
$$F <function name>	defines a function name for a function other than the target that is included in `test.c`
$$T <function name>	defines the function name that is the target (i.e. called function) of the mutation process, included in `test.c`

Procedure

Assume the function to be modified is named "`function_1`". The general procedural outline for using the programs listed above is as follows.

1. cut function `function_1` to be mutated out of the file it is in and put into a file named `test.c`
2. remove all comments from the function `function_1`, the mutation tool cannot currently deal with comments as they can run over multiple lines.
3. ensure all statements are on a single line, otherwise the mutation tool can't cleanly delete the statement.
4. ensure that any called functions that are also being mutated appear *before* the function under test. Please note that the ability to modify called functions has only been tested with a small number of examples.
5. Modify the function `function_1` (and other functions) to ensure that the tool can recognize it (see examples) by separating the function name from any parameters it may have.
6. Use special tags to define, at the start of the file, in the following order
   a) the names of global and static scalar and array variables using the `$$V` and `$$A` tags.
   b) the names of any global functions being called using the `$$N` tag.
   c) the names of any functions called by the function that are also to be mutated using the `$$F` tag.
   d) the name of the (one) target function using the `$$T` tag, the target function is the function that will be called in the test harness.

Note : the $$F and $$T tags must follow the $$V and $$A tags and the $$T tag *must* be the last in the list. If there are multiple functions in the `test.c` file and this is not done then `line.c` may get confused about line number ranges for the different functions. This can be checked in `symbol_table.txt` and may be flagged by the compiler as large numbers of mutants that will not compile.

7. add the function definition as it appears in text.c to the file nogen.txt followed by a '$' character at the start of the next line. At this point no mutant number should follow the '$' character as we don't know what mutants we want to suppress.
8. Use the mutation program `line.c` to mutate the code in `test.c`, pipe debug information to a separate text file e.g.

```
a.out >res.txt
```

Note : the majority of the printf debugging statements have been left in the code and the file res.txt can be quite large. The program runs very slowly if output isn't piped to a file.

9. Check the file `symbol_table.txt` for errors.
10. Modify the output file pointers.h by filling in the correct function definition and removing the last comma from the initializer.
11. customize the file o`racle.c` as described below and as shown in the examples.
12. modify `driver.c` (if necessary) as described in the section below and as shown in the examples.
13. compile `driver.c` note the function numbers of all the mutants that compile with errors (warning can be ignored if the user so chooses). Because of the huge number of errors that will occur the following command can be used under Linux to pipe the warnings and errors to a text file.

```
gcc driver.c >comp.txt 2>&1
```

14. Mutant functions that do not compile can be dealt with by removing the body of the code and replacing it with the following code, this causes a runtime failure which `driver.c` can deal with on it's own. Note that the whole function cannot be deleted as this would alter the sequencing of the function names in `pointers.h` file.

    ```
    int i, j; i = 120; j = 120; i = i / (j - i); return;
    ```

15. Step 14 can either be done automatically by adding the information on what mutants to suppress to the `nogen.txt` file or it can be done the old fashioned way and the `mutant.c` file can be edited by hand (very, very, very time consuming).
16. repeat steps 13 and 14 until the program has compiled.
17. Run the compiled version of the driver program, piping the standard output into a text file, e.g.

    ```
    a.out >res.txt
    ```

The nogen file

The file nogen.txt is used to suppress the generation of specific numbered mutants. The file itself has the format, target function definition, a '$' separator on a line by itself, the list of mutant numbers to suppress, one per line.

For example the following is the initial `nogen.txt` file for the `heapSort` example. It has five lines, the first three are the function definition of the target function from `test.c` and the fourth is the separator line. The last line is blank, this is where the numbers to suppress will go.

```
void heapSort(
    int numbers[],
    int array_size)
$
```

After the `driver.c` program has been compiled for the first (second, third etc.) time a set of mutants that cannot be compiled will be known, these are added after the separator line. For example if the mutants 5, 9, 23,and 47 produced compilation errors then the file would become;

```
void heapSort(
    int numbers[],
    int array_size)
$
5
9
23
47
```

Running `line.c` with this version of `nogen.txt` will force the mutants numbers to generate code that forces a divide by zero.

The oracle file

This file is used to define specific instances of functions called by the driver that run, first the oracle and then the target mutation.

1. Add global and static data to the global data section, names should be modified to differentiate them from actual variable names by (for example) adding "xyz_" before the name. All data should be an array of two elements, one to record the oracle results and one for the results from the mutant function.

   example : if the global integer scalar `g_loop_executions` exits then a variable `xyz_g_loop_execution [2]` should be created.

   A hint on renaming variables that are structures or arrays to make life simple during this step is to convert "." to "_" and for arrays convert "[" to "_" and "]" to either "_" or "".

   example : `cat.dog[2].mouse[4]` becomes `cat_dog_2_mouse_4`
2. If the function that has been mutated calls functions that are not mutated then dummy functions can be defined here to record parameters passed, number of calls etc. and pass back values defined in the test vector.

3. In `drive_function` set the global, static and parameter data input values to the values in the vector array defined in `test_vectors.h`
4. In `drive_function` modify the call to `*target_array[0]` (the oracle function) and `*target_array[g_target_fn]` (the mutated function) to include the correct parameters and collect the return value if there is one.
5. At the end of `drive_function` all returned values need to be collected from the global data into the variables defined in step 1. The array index used is "index" which is passed to the function and defines whether the data has been collected from the oracle (index = 0) or a mutant (index = 1).
6. In the function `drive_compare` values defined in step one are compared against each other, this function has the following format (this is a truncated example of real code).

```
int drive_compare (void)
{
    int i;
    int ok = 1;
    if (xyz_gov_tq [0]           != xyz_gov_tq [1]) ok = 0;
    if (xyz__gov_ffd_active [0]  != xyz__gov_ffd_active [1]) ok = 0;
    if (xyz__gov_ffd_rpm [0]     != xyz__gov_ffd_rpm [1]) ok = 0;
    return (ok);

} /* end drive-compare */
```

The driver file

Several modifications may be necessary to the `driver.c` file to allow compilation of the code under test. The modifications that have been used to date include the following;

- inclusion of any header files required by the code under test at the start of the file.
- definition of global and static data, enumerations etc. that are defined in the file that the function under test has been extracted from. These are included in the section labeled "Local Variable from test file".

The test vector file

The file that contains the test vectors has a specific format and contains information used by `driver.c` on the number of test vectors and the number of elements in each vector. A simplified example is shown below.

```
int a01 [] = {    1450,     3924,      532,       19};
int a02 [] = {    3407,      824,     1460,       19};
int a03 [] = {    4032,     2189,     2574,       19};
/* snipped */
int a26 [] = {    3641,     1030,     3174,       19};
int a27 [] = {    3710,     2375,     2448,       19};

int v_num_vectors = 27;
int v_size_vector = 4;

/* pointer array to the test data */
int *v_testdata [] = {
a01, a02, a03, a04, a05, a06, a07, a08, a09, a10,
a11, a12, a13, a14, a15, a16, a17, a18, a19, a20,
a21, a22, a23, a24, a25, a27, a27
};
```

The first section defines the test vectors `a01` though `a27,` following this is the definition of `v_num_vectors` which defines the number of test vectors and `v_size_vector` which defines the size of the test vectors. Finally we have the definition of a static pointer array which `driver.c` uses to access the actual data.

Note that the format has been defined to allow the Csaw tool to work with automated test data generation programs the author is working with.

Worked Example 1 – Bubble sort

The original file for bubble sort looks something like the following.

```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;  /* working storage */
  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

This is modified to become the following in the file `test.c`, note the use of the `$$T` tag at the start of the file and the way in which the function argument list has been split onto a separate lines. This is necessary to allow the function to be matched against a prototype string of the form "`void <function identifier>  (`" defined in the tables.

```
$$T bubbleSort

void bubbleSort (
    int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

At the same time the following is put into the file `nogen.txt` file.

```
void bubbleSort (
    int numbers[], int array_size)
$
```

The program `line.c` has its debugging comments intact and so produces a flood of data that unless redirected to a text file and significantly slows down the program. Redirection is strongly recommended! Also produced is a set of symbol tables in the file `symbol_table.txt` which should be examined for errors. The last two output tables of the symbol table file are shown below, the first is for all the strings the tool has decided are variable names, the second is all the strings it has decided are constants. For `bubbleSort` there are no constants. Other versions of the table are output in front of these two and are for debugging purposes.

The symbol tables include the following information for each of the symbols listed within them.

- The string "ST" is just a marker that could be searched for when symbol tables were output as part of the general output stream.
- The first number is the row number in the symbol table (a large array) where the data is held.
- The second number is the line number (row in array) where the symbol was defined.
- The character 's' indicates that the name is "special" i.e. it was derived from a tag, if the symbol is not special then this is blank.
- The range of numbers e.g. "`2-19`" is the range of lines in the function over which the symbol is valid, for a single function like `bubbleSort` these should all be the same.
- The next string e.g. "`target`" or "`int`" is the base type of the symbol, `target` indicates it is the target name for the function, `int` that it is an integer (whether unsigned, short or other).

- The next string is the kind of symbol it is, options are `funct` (function) `scalar` or `array`. Note that for this field and the previous one, if it is none of the valid options then “ERROR” is printed. In the case of constants “ERROR” is always output.
- The next filed is the length of the symbol in characters.
- The last field is the actual text of the symbol delimited by '>' and '<'.

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    SYMBOL   TABLE VARIABLES
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                   lines                len     text
ST   0 :   0 :s:   2- 19 :target funct   10 >bubbleSort<
ST   1 :   3 : :   2- 19 :int    array    7 >numbers<
ST   2 :   3 : :   2- 19 :int    array   10 >array_size<
ST   3 :   5 : :   2- 19 :int    scalar   1 >i<
ST   4 :   5 : :   2- 19 :int    scalar   1 >j<
ST   5 :   5 : :   2- 19 :int    scalar   4 >temp<

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    SYMBOL   TABLE CONSTANTS
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                   lines                len     text
```

The `line.c` program also produces the `mutant.c` file with the following format. Each copy of the function has had its name altered, the original function and oracle becomes `bubbleSort0000` and the first mutant is `bubbleSort0001`, the second `bubbleSort0002` and so on. Also notice that the point where the file is mutated is marked with a comment which also gives the type of the mutation, which for `bubbleSort0001` is “`type mutation`”. The heading comment for each function also includes its number and the number of the lines that was mutated.

```
/********** original code 0000  **********/

void bubbleSort0000 (
    int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}

/***** next mutation 0001 for line   3 *****/

void bubbleSort0001 (
    unsigned int numbers[], int array_size) /* type mutation */
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

The program `line.c` also produces a table of pointers to the function, a section of which is shown below. Where the comment "`/* parameters here */`" occurs the actual parameters for the function need to be substituted (see below).

```
typedef void(*ptr2funct)(/* parameters here */);

ptr2funct target_array [] =
{
    bubbleSort0000,
    bubbleSort0001,
    bubbleSort0002,
    bubbleSort0003,
    bubbleSort0004,
    bubbleSort0005,
    bubbleSort0006,
    bubbleSort0007,
```

Replacing the parameter list gives us the following.

```
typedef void(*ptr2funct)(int numbers[], int array_size);
```

Before the oracle and driver can be constructed the format of the input data needs to be known. In this case there are eight input values, the first is the "size" or the array to be sorted, a value from 1 to 7 and the remainder are the array elements from one to seven. Information on the number of vectors and the size of each vector is provided to `driver.c` in two global variables `v_num_vectors` and `v_size_vector`. An array of pointers `v_testdata` is also used so `driver.c` can locate the each array.

```
int a001 [] = {3, 83984096, 11401506, ..., 17641812, 31246168, 49272653};
int a002 [] = {7, 46854651, 55085516, ..., 75111624, 61627822, 28944913};
    /* snip */
int a010 [] = {3, 18466108, 60017092, ..., 42777661, 42474554, 66665136};


int v_num_vectors =  10;
int v_size_vector =   8;

/* pointer array to the test data */
int *v_testdata [] = {
a001, a002, a003, a004, a005, a006, a007, a008, a009, a010
};
```

The driver program for bubbleSort has been included in it's entirety below. First are a number of global data variables as defined previously, `dummy` is used as local shortage for the array to be stored and `results_1` is used to store returned data for the oracle and mutant so it can be compare latter.

The `drive_function` function comprises two parts, the first to initialise and copy data to working variables declared above and the second to perform the calls to the oracle and mutant functions and record the results of the calls.

The function `drive_compare` compares the results of the calls made to the oracle and mutant functions and indicates whether the results were the same (mutant is alive) or if they differed (mutant is dead).

```
/*
 * global data values that are used to record the execution result of
 * the functions under test
 */

int g_update_index;  /* do NOT remove */

int dummy [10];
int results_1 [2][10];

/****** Dummy procedures go here *******/

/*
 * drive_function : is used to call the function under test, it has
 * two parts...
 */
void drive_function (int index, int vector [])
{
    int i, size;
    int dummy_value;

    /* set global to a known value */
    for (i = 0; i < 10; i++)
```

```
            dummy[i] = 78787878;   /* default value */
        size = vector[0];
        for (i = 0; i < size; i++)
            dummy[i] = vector[i+1];

        /* set up the global index for called functions (if ant) */
         g_update_index = index;
        /* call the target function e.g. */
        /* _dip_check_cal_return = _dip_check_cal (p1,p2); */
        if (index == 0)
        {
            /* call the oracle - always first in array */
            (*target_array[0])(dummy, size);
            /* record global data - if necessary */
        }
        else if (index == 1)
        {
            /* call the mutant */
            (*target_array[g_target_fn])(dummy, size);
        }
        else
        {
            printf ("ERROR : bad index for mutant call\n");
        }

        /* record global data - if necessary */
        for (i = 0; i < size; i++)
            results_1[index][i] = dummy[i];

    } /* end drive_function */

    /*
     * drive_compare : is a bespoke function used to compare the results
     * from either the oracle (index 0) or the mutant (index 1) any
     * difference returns a failure
     */

    int drive_compare (void)
    {
        int i;
        int ok = 1;

        for (i = 0; i < 10; i++)  /* for each element */
            if (results_1[0][i] != results_1[1][i]) ok = 0;

        return (ok);

    } /* end drive-compare */
```

The `driver.c` program is compiled in the usual manner and output is directed to the terminal (using `printf`) so it need to be redirected to a text file. The test results for running the driver above are shown below.

The first section comprises output data for each of the first five tests executed (Test ID 0 to 4) and gives the following information. TESTFAIL is the number of failures detected, where the output from the mutant does not match the oracle results. The TESTORAC is now redundant it was used during development to check that the oracle was called correctly. TESTPASS is the number of mutants that passed, i.e. produced output that cannot be distinguished from the oracle. TESTCRASH is the number of mutants that caused an exception e.g. divide by zero. TESTKILL is the number of mutants that needed to be killed because they exceeded the allowed time limit so have possibly entered an infinite loop and TESTERR is reported for any unrecognised return value or if a `fork` system call failed. KILLED is the sum of TESTFAIL, TESTCRASH and TESTKILL and Total is the total number of mutants exercised.

Following this is a summary of the number of vectors and their size (number of elements) and the number of mutants exercised. This is followed by a list of all the mutants which survived all tests and a list of how many mutants each test vector killed. After this comes a list of mutants killed by only a single test vector (empty here) and then the elapsed clock time.

```
Test ID =    0
parent : TESTFAIL  =   90
parent : TESTORAC  =    0
parent : TESTPASS  =   32
parent : TESTCRASH =   24
parent : TESTKILL  =   11
parent : TESTERR   =  158

parent : KILLED    =  125
parent : Total     =  157

Test ID =    1
parent : TESTFAIL  =   98
parent : TESTORAC  =    0
parent : TESTPASS  =   24
parent : TESTCRASH =   24
parent : TESTKILL  =   11
parent : TESTERR   =    0

parent : KILLED    =  133
parent : Total     =  157

Test ID =    2
parent : TESTFAIL  =   98
parent : TESTORAC  =    0
parent : TESTPASS  =   24
parent : TESTCRASH =   24
parent : TESTKILL  =   11
parent : TESTERR   =    2

parent : KILLED    =  133
parent : Total     =  157

Test ID =    3
parent : TESTFAIL  =   93
parent : TESTORAC  =    0
parent : TESTPASS  =   29
parent : TESTCRASH =   24
parent : TESTKILL  =   11
parent : TESTERR   =    0

parent : KILLED    =  128
parent : Total     =  157

Test ID =    4
parent : TESTFAIL  =    7
parent : TESTORAC  =    0
parent : TESTPASS  =  123
parent : TESTCRASH =   18
parent : TESTKILL  =    9
parent : TESTERR   =    0

parent : KILLED    =   34
parent : Total     =  157
===============================================================
Number of vectors =   10
Size of vector    =    8
Number of mutants =  158 ( 157)
===============================================================

Immortal mutants that passed all tests are...
    1    2    4    7    8    9   10   11   12   14   16
   25   31   34   40   41   57   58   60   83
End Immortal list....   20 not killed

vector   0 killed  125 mutants
vector   1 killed  133 mutants
vector   2 killed  133 mutants
vector   3 killed  128 mutants
vector   4 killed   34 mutants
vector   5 killed  116 mutants
vector   6 killed  113 mutants
vector   7 killed  137 mutants
vector   8 killed  133 mutants
vector   9 killed   36 mutants

Elasped time =        232 seconds
```

The file bit_log.txt displays a matrix of test vectors (rows) vs. mutants (columns) with an '.' for a live mutant and an 'X' for a dead mutant. The initial columns of this matrix are shown below.

```
  0:..X.XX......X.X.XXXXXXXX.XXXXX.XX.XXXXX..XXXXXXXXXX.X.......XXXXXXX.X
  1:..X.XX......X.X.XXXXXXXX.X...X..X.XXXXX..XXXXXXXXXXXXXXX..X.XXXXXXXXX
  2:..X.XX......X.X.XXXXXXXX.X...X..X.XXXXX..XXXXXXXXXXXXXXX..X.XXXXXXXXX
  3:..X.XX......X.X.XXXXXXXX.XXXXX.XX.XXXXX..XXXXXXXXXX..........XXXXXX.X
  4:....X.......X...X........................XX..XXXXX.............XXXX.X
  5:..X.XX......X.X.XXX.XXXX.X...X....XXXXX..XXXXXXXXXX...........XXXXX.X
  6:..X.XX......X.X.XXXXXXXX.XXXXX.XX.XXXXX..XXXXXXXXXX.X.......XX.XXXX.X
  7:..X.XX......X.X.XXXXXXXX.XXXXX.XX.XXXXX..XXXXXXXXXXXXXXX..X.XXXXXXXXX
  8:..X.XX......X.X.XXXXXXXX.X...X..X.XXXXX..XXXXXXXXXXXXXXX..X.XXXXXXXXX
  9:..X.XX......X...X........................XX..XXXXX.............XXXX.X
```

Worked Example 2 – Heap Sort

This shows an example where there are two functions being passed to `line.c` one `shiftDown` being called from the target function `heapSort`. Two things to be noted, the called function *must* appear before the function that calls it and the tags must be declared in the order shown. The symbol table building functions of `line.c` get confused otherwise. This shows up in the mutated program text as invalid variable (and constant) substitutions. For example `root` from `shiftDown` being substituted for `temp` in the `heapSort` function.

```
$$F siftDown
$$T heapSort

void siftDown(
     int numbers[],
     int root,
     int bottom)
{
  int done, maxChild, temp;

  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
     /* snip */
  }
}

void heapSort(
    int numbers[],
    int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);

  for (i = array_size-1; i >= 1; i--)
  {
       /* snip */
    siftDown(numbers, 0, i-1);
  }
}
```

Note that in these functions both names are modified in all places that they occur. So all instances of `heapSort` become `heapSortxxxx` and all instances of `shiftDown` becomes `shiftdownxxxx`.

```
void siftDown0010(
     int numbers[],
     long int root, /* type mutation */
     int bottom)
{
  int done, maxChild, temp;

  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;

    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}

void heapSort0010(
    int numbers[],
    int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown0010(numbers, i, array_size);
```

```
      for (i = array_size-1; i >= 1; i--)
      {
        temp = numbers[0];
        numbers[0] = numbers[i];
        numbers[i] = temp;
        siftDown0010(numbers, 0, i-1);
      }
    }
```

Worked Example 3 – Median Filter

This shows an example where there are two global variables are used in a function (fairly common in embedded code for a number of reasons). Both are arrays and are defined to the `line.c` program *before* the target function is tagged.

```
$$A history_1
$$A history_2
$$T median_filter

static unsigned short int median_filter (
     enum INPUT_NAMES_T logical_id,
     unsigned short int value )
{
    unsigned short int values [3];
    unsigned short int tmp;
    unsigned char  smallest;

    values [0] = value;
    values [1] = history_1 [logical_id];
    values [2] = history_2 [logical_id];
    smallest = 0;

    /* snip */
}
```

The two global arrays need to be declared in the `driver.c` program as shown in the following section of `driver.c`.

```
/*
 ***********************************************************************
 *   Local Variable from test file
 ***********************************************************************
 */

static unsigned short int history_1 [NUM_INPUT_NAMES + 1];
static unsigned short int history_2 [NUM_INPUT_NAMES + 1];

/* header files for the mutants under test */
#include "mutant.c"
#include "pointers.h"
```

Acknowledgments

Special thanks to Dennis M. Rictchie for not using the '$' character as an operator in C, it made the construction of this tool much simpler. To Jeff Offutt for informing me that there were no C mutation tools available and to James Andrews for supplying a version of his Prolog based mutation tool which acted as inspiration for the construction of this tool.

References

1. Agrawal, H., DeMillo, R.A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A.P. and Spafford, E. Design of Mutant Operators for the C Programming Language, Department of Computer Science, Purdue University, W. Lafayette, 2006, 1-74.
2. Andrews, J.H., Briand, L.C. and Labiche, Y. Is mutation an appropriate tool for test experiments? *Proceedings of the 27th international conference on Software engineering*, ACM Press, St. Louis, MO, USA, 2005.
3. Andrews, J.H., Briand, L.C., Labiche, Y. and Namin, A.S. Using Mutation Analysis for Assessing and Comparing Test Coverage Criteria. *IEE Tran. Soft. Eng.*, *32* (8). 608-624.

4. Barbosa, E.F., Maldonado, J.C. and Vincenzi, A.M.R. Toward the Determination of Sufficient Mutant Operators for C. *Software Testing Verification And Reliability Journal*, *11*. 113 - 136.
5. Bentley, J. Programming Perls: Bumper-Sticker Computer Science. *Com. ACM*, *28* (9).
6. Cohen, D.M., Dalal, S.R., Fredman, M.L. and Patton, G.C. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. Softw. Eng.*, *23* (7). 437-444.
7. Delamaro, M.E. and Maldonado, J.C., Proteum - A tool for the assessment of test adequacy for C programs. in *Proceedings of the Conference on Performability in Computing Systems*, (1996), 79 - 95.
8. Delamaro, M.E., Maldonado, J.C. and Mathur, A. Interface Mutation: An Approach for Integration Testing. *IEEE Transaction On Software Engineering*, *27*. 228 – 247.
9. Delamaro, M.E., Maldonado, J.C., Pasquini, A. and Mathur, A. Interface Mutation Test Adequacy Criterion: An Empirical Evaluation. *Journal Of Empirical Software Engineering*, *6*. 111 - 142.
10. Delamaro, M.E., Maldonado, J.C. and Vincenzi, A.M.R., Proteum/IM 2.0: An Integrated Mutation Testing Environment. in *Mutation 2000: A Symposium on Mutation Testing for the New Century*, (San Jose, 2000), 124 – 134.
11. DeMillo, R.A., Lipton, R.J. and Sayward, F.G. Hints on test data selection: help for the practising programmer. *Computer*. 34-41.
12. DeMillo, R.A. and Offutt, A.J. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.*, *17* (9). 900-910.
13. DeMillo, R.A. and Offutt, A.J. Experimental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol.*, *2* (2). 109-127.
14. Ellims, M., Ince, D. and Petre, M., The Csaw C Mutation Tool: Initial Results. in *Mutation Analysis 2007*, (Windsor, UK, 2007), IEEE.
15. Hamlet, R.G. Testing Programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, *3* (4). 279-290.
16. King, K.N. and Offutt, A.J. A Fortran language system for mutation-based software testing. *Softw. Pract. Exper.*, *21* (7). 685-718.
17. Maldonado, J.C. and Barbosa, E.F., Establishing a Mutation Testing Educational Module based on IMA-CID I. in *2nd Workshop on Mutation Analysis (Mutation 2006)*, (2006), 1 – 1.
18. Maldonado, J.C., Delamaro, M.E., Fabbri, S.C.P.F., Simão, A.S., Sugeta, T., VincenzI, A.M.R. and Masiero, P.C., Proteum: a Family of Tools to Support Specification and Program Testing Based on Mutation. in *Mutation 2000: A Symposium on Mutation Testing for the New Century*, (San Jose., 2000), 146 - 149.
19. Mathur, A.P. Performance, effectivness and reliability issues in software testing *Proc. of the 15th Annual International Computer Software and Applications Conference*, Tokyo, Japan, 1991.
20. Namin, S.A. and Andrews, J.H., Finding Sufficient Mutation Operators via Variable Reduction. in *Second Workshop on Mutation Analysis*, (2006), IEEE.
21. Offutt, A.J. A Practical System for Mutation Testing: Help for the Common Programmer *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*, IEEE Computer Society, 1994.
22. Offutt, A.J., Gregg, R. and Christian, Z. An experimental evaluation of selective mutation *Proceedings of the 15th international conference on Software Engineering*, IEEE Computer Society Press, Baltimore, Maryland, United States, 1993.
23. Offutt, A.J. and King, K.N. A Fortran 77 interpreter for mutation analysis *Papers of the Symposium on Interpreters and interpretive techniques*, ACM Press, St. Paul, Minnesota, United States, 1987.
24. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H. and Zapf, C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, *5* (2). 99-118.
25. Offutt, A.J. and Voas, J.M. Subsumption of condition coverage techniques by mutation testing *Tech. Report*, Dept. of Information and Software Systems Engineering , George Mason Univ., Fairfax, Va., 1996.
26. Offutt, J. and Seaman, E.J. An Integrated automatic test data generation system *Proceedings of the Fifth Annual Conference on Systems Integrity, Software Safety and Process Security (COMPASS '90)*, Gaithersburg, MD, USA, 1990.
27. VincenzI, A.M.R., Maldonado, J.C., Barbosa, E.F. and Delamaro, M.E. Unit and Integration Testing Strategies for C Programs Using Mutation-Based Criteria. *Software Testing Verification And Reliability Journal*, *11*.
28. Vincenzi, A.M.R., Nakagawa, E.Y., Maldonado, J.C., Delamaro, M.E. and Sanches, R. Bayesian-learning based guidelines to determine equivalents mutants. *International Journal Of Software Engineering And Knowledge Engineering*, *12*. 1 – 15.
29. Wong, W.E., Horgan, J.R., Mathur, A.P. and Pasquini, A. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application *Proceedings of the 21st International Computer Software and Applications Conference*, IEEE Computer Society, 1997.