2005－SE－147（7）
2005／3／17

ミューテーション法を用いたテストセット構成支援に関する研究

斉藤 孝志　　大久保 弘崇　　粕谷 英人　　山本 晋一郎

愛知県立大学大学院 情報科学研究科

要 旨

DeMillo らの提案したミューテーションスコアはテストセットの品質の尺度である．本稿では，与えられたテストセットから，ミューテーションスコアを下げない最小の部分集合を抽出することで，テストセットの構成を支援する方法を提案する．抽出された部分テストセットは，初期テストセットによるテストとミューテーションスコアの観点から同等の効果がある．我々の方法は，作成コストの高い正解出力を必要としない．すなわち，本提案手法によりテストセットを構成することで，正解出力作成コスト・テスト実行コスト・バグの検出能力の 3 つの側面から効果的なテストが行える．

gzip-1.2.4 に対する評価実験で，初期テストセット (要素数 2326 個, 命令網羅率 76%, ミューテーションスコア 63%) から，要素数 29 個の最小テストセット (命令網羅率 75%, ミューテーションスコア 63%) を抽出した．

Test Set Construction Method Based on Mutation Testing

Takashi SAITO, Hirotaka OHKUBO, Hideto KASUYA, and Shinichiro YAMAMOTO

Graduate School of Information Science and Technology, Aichi Prefectural University

Abstract

DeMillo proposed the mutation score to measure the quality of a test set. In this paper, we propose a test set construction method by extracting the smallest subset from a given test set, where the mutation score of the extracted subset is eqaul to that of given one. Thus the extracted subset has the same effectiveness as the given test set in the sense of mutation score. Our method does not require the output part of the test set which take high cost. In short, it enables to test effectively on three sides: the correct output data making cost, test's executing time and bug detecting effeciency.

By experimental implementation, we evaluated the method on gzip-1.2.4. From the initial test set that consists of 2326 input data, the smallest test set that consists of 29 input data has extracted.

1　はじめに

ソフトウェアの開発において，テストの工程にかかる労力は大きい．発見されたバグの修正はソフトウェア開発の様々な工程へ影響を与える可能性がある．開発期間は限られているため，ある一定期間内のテストによりソフトウェアの信頼性を保証しなければならない．

プログラムのテストは，対象プログラムへ入力を与えた時にその出力が期待されたものであるかどうかを確認することで行う．そこで用いる**入力データ**とその入力データに対して期待される出力である**正解出力**の一対を**テストケース**と呼ぶ．プログラムの検査項目に応じたテストケースを組み合わせた**テストセット**によりテストを行う．ある入力データに対する対象プログラムの出力が正解出力と異なるとき，対象プログラム中に何らかのバグがあることを確認できる．可能な限り多くのテストケースでテストすることにより，対象プログラム内の様々なバグを検出することが期待される．

しかし，入力データに対する正解出力は，対象プログラムによる算出以外の手段で得る必要があるため，その作成には多大なコストを要する．ソフトウェアの開発における統合テストは，各モジュールを統合したプログラム全体のテストであり，各モジュールの相互作用によりプログラムの振る舞いは複雑化する．このため，テスト項目も増え，そのテストセットの作成は多大な労力を要する．従って，少ない数のテストケースで様々なバグを検出することができるテストセットの構成法が求められている．

テストセットのバグ検出能力を測定する方法として，DeMillo らはミューテーション法を提唱した [1]．ミューテーションとは故意にプログラムへ微少な変化を加えることである．ミューテーション法は，テストセットのバグ検出能力が高ければ故意に挿入したバグであるミュータントをテストで検出できるだろうという考えに基づいている．ミュータント群に対して，テストセットがミュータントを検出した割合を**ミューテーションスコア**と呼ぶ．このミューテーションスコアがテストセットのバグ検出能力の指標となる．

ミューテーション法は，テストセット全体のバグ検出能力の尺度であるだけでなく，個々のテストケースの重要度の尺度にも使える．この視点に基づき，対象プログラムの大量の入力データ群から，ミューテーションスコアの観点において有用なものだけを選択する方法を提案する．ここで選択した入力データの集合に対して正解出力を作成すれば，正解出力作成コスト，テスト実行コスト，バグ検出能力の 3 面で効率的なテストが行える．

2　ミューテーション法

2.1　ミューテーション法の概略

ミューテーション法は，プログラムのテストに用いられるテストセットの品質を評価する方法である．プログラムへ故意に微少な変更である**ミューテーション**を行い，テストセットがその変更点を検出できるかどうかを調べる．この微少な変更点を特に**ミュータント**，ミュータントを含むプログラムのことを**ミュータントプログラム**と呼ぶ．ミュータントプログラムに対し，元のプログラムを特に**オリジナルプログラム**と呼ぶ．

ある入力データについてあるミュータントプログラムの結果が正解出力と異なるとき，ミュータントが検出されたことを表す．このことを特にミュータントの**摘出**と呼ぶ．ミュータントはオリジナルプログラムから複数作成されるが，各ミュータントプログラムに含まれるミュータントの数は常に 1 つとしている．これは，複数のミュータントを含むことでの大きな利点が望めないためである [2]．

そして，テストセットがオリジナルプログラムから生成された複数のミュータントをどれだけ摘出できるかをミューテーションスコアとして表す．このミューテーションスコアの数値が，テストセットのバグ検出能力の指標である．

2.2　ミュータントオペレータ

ミュータントはオリジナルプログラムからある規則に従っていくつも生成される．この規則がミュータントオペレータである．ミュータントオペレータはソースプログラムのステートメントや変数などを対象要素として，その対象要素へどのようなミューテーションを行うかを定めている．ミュータントオペレータは，プログラマが犯しやすい間違いのモデルである．

Fortran と C 言語のミュータントオペレータが，それぞれ King ら [3]，Agrawal ら [4] によって定義されている．本稿は C 言語プログラムを対象とするため，Agrawal らによって定義されているミュータントオペレータの一部を利用する．

2.3 ミューテーションスコア

ミューテーション法においてテストセットの品質を示す指標がミューテーションスコアである．オリジナルプログラムから複数のミュータントオペレータを適用することにより生成されたミュータントの総数と，摘出されたミュータントの数からミューテーションスコアを算出する．

オリジナルプログラムを P，そのテストセットを T とする．そして，P から生成されたミュータントの数を M，摘出されたミュータントの数を K，等価ミュータントの数を E とする．等価ミュータントとはオリジナルプログラムと関数的に同一の効果を示すミュータントである．そのため，ミューテーションスコアを計算する際にはその数を除外する．以上より，ミューテーションスコア $MS(P,T)$ の定義は式 (1) である．

$$MS(P,T) = \frac{K}{M-E} \times 100 \quad [\%] \qquad (1)$$

3 テストセット構成法

3.1 テストセット構成法の指針

本稿が提案する手法は，テスト対象プログラムのどの入力データをテストケースとすべきかの選択を支援するものである．1 章で述べたとおり，正解出力の作成にかかるコストを考えると，テストケースの数は小さい程よい．さらに，そのテストセットが高いバグ検出能力を持っていることにより，効果的なテストが実施できるといえる．

本稿ではテストセットのバグ検出能力を 2 章で述べたミューテーションスコアとして捉える．ミューテーション法はテストセット全体のバグ検出能力の指標であるミューテーションスコアを算出する過程で個々の入力データがどのミュータントを摘出するかを調べている．すなわち，個々の入力データについてもミューテーションスコアを算出することが可能であり，それは個々のテストケースの重要度の尺度として利用できる．

個々のテストケースについてのミューテーションスコアに注目すると，与えられた入力データ集合と等しいミューテーションスコアを持つ最小の部分集合を考えることができる．バグ検出能力をミューテーションスコアで近似したとき，この部分集合から成るテストセットは初期のテストセットと等しいバグ検出能力を持ち，正解出力作成コストが最小になるテストセットであるといえる．以下，入力データの集合から以上の観点に基づく最小の部分集合を求める手続きを提案する．

3.2 摘出行列

与えられた入力セットから必要な入力データを選別するために個々の入力データがどのミュータントを摘出するか否かを考える必要がある．この情報を表す方法として，摘出行列を定義する．

摘出行列とは，オリジナルプログラムから生成された M 個の各ミュータントについて，N 個の入力データがそれらを摘出できるかどうかを $M \times N$ の行列として表したものである．摘出行列の各入力データを各列に，各ミュータントを各行に対応させる．この摘出行列 $\boldsymbol{K}$ の作成方法を図 1 に示し，以下で説明する．

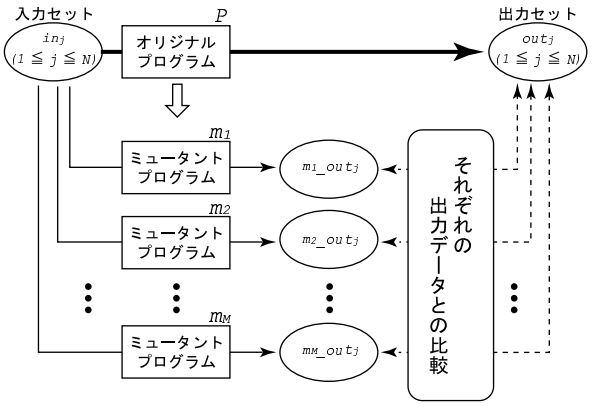

図 1: 摘出行列の作成

1. 各入力データに対するオリジナルプログラム P の出力結果を求める．
   出力結果は，各入力データに対するミュータントプログラムの出力と比較するために用いる．入力データ in_j $(1 \le j \le N)$ に対する出力データを out_j $(1 \le j \le N)$ とする．
2. 各入力データに対する各ミュータントプログラムの出力を求める．
   ミュータント生成システムで生成されたミ

ュータント m_i $(1 \leq i \leq M)$ を含むミュータントプログラム P_i を各入力データ in_j で実行する．入力データ in_j に対するミュータントプログラムの出力データを $m_i\_out_j$ $(1 \leq i \leq M,\ 1 \leq j \leq N)$ とする．

3. 摘出行列の作成
入力データ in_j がミュータント m_i を摘出したとき，すなわち，$out_j \neq m_i\_out_j$ のとき $\boldsymbol{K}$ の i,j 要素 $k_{i,j}$ の値を 1 とし，摘出できない，すなわち，$out_j = m_i\_out_j$ のとき $k_{i,j}$ を 0 とする．

2 章では，入力データに対してオリジナルプログラムの出力と正解出力を比較することによってミュータントを摘出できるかどうかを調べている．しかし，本手法ではテストケース候補となるべき入力データを選択するため，その入力データに対するオリジナルプログラムとミュータントプログラムの出力を比較することによってミュータントが摘出されるかどうかを調べる．ミュータントを摘出できる入力データは，そのミュータントを摘出するという確かなバグ検出能力を保持している．従って，入力データとそれに対する正解出力からなるテストケースは，何らかのバグ検出能力を持っているといえる．

各入力データがどのミュータントを摘出するかどうかを調べるために，オリジナルプログラムとミュータントプログラムの出力結果を比較する．そのために，各ミュータントプログラムを各入力データで実行して出力結果を得る必要がある．しかし，全てのミュータントプログラムが終了するものとは限らない．無限ループに陥るミュータントプログラムが生成される可能性がある．このように，一定時間経過しても終了しないミュータントも摘出されたものとして扱う．

各入力データが次の 2 つの条件のどちらかを満たしたとき，ミュータントを摘出したものとする．

- プログラムが終了したときに，その出力結果がオリジナルプログラムの結果と異なる．
- 一定時間経過してもプログラムが終了していない．ミュータントプログラムが一定時間経過した後，オリジナルプログラムと同じ結果を出力することがあるかもしれない．これは，ミュータントプログラムが終了するまでオリジナルプログラムとは異なる振る舞いをしているといえる．

3.3 テストセット構成手法

本手法では，テスト対象プログラムへの N 個の入力データを持つ入力セットから，ミューテーションスコアを下げない最小の部分集合を求める．これは，元の入力セットが摘出できる全てのミュータントを最低 1 つの入力データが摘出する最小の集合を求める，最小被覆問題に帰着できる．

4 評価実験

3 章で提案したテストセット構成法を用いて，実際に gzip-1.2.4 (以下 gzip) のテストセットを構成する．

4.1 gzip

gzip (GNU zip) はファイルの圧縮・伸張をするオープンソースのソフトウェアで，UNIX 系 OS で広く普及している．

gzip が提供する機能は，次の 4 種に分類できる．

F1: ファイルの圧縮

F2: 圧縮ファイルの伸張

F3: 圧縮ファイルに関する数種類の情報の提示

F4: 圧縮ファイルの整合性の検証

4.2 実験の手順

本研究で提案するテストセット構成法を用いて gzip プログラムに対するテストデータ群を構成した．この評価実験は，以下の手順で行った．

1. 入力セットの作成
2. ミュータントの生成
3. ミュータントプログラムのテスト
4. テストセットの構成．

評価実験は，表 1 の環境の元で行った．

表 1: 評価実験の環境

Sun Blade 2000

OS	Sun OS 5.9
CPU	UltraSPARC-3 Cu 1.2GHz
メモリ	1.0GB

4.2.1 入力セットの作成

評価実験を行うにあたり，2326 個の入力データからなる入力セットを作成した．これらの入力データは，4.1 節で分類した gzip の各機能 (F1〜F4) をテストするために作成した．また，その他のものとして，ヘルプの表示や異常終了を想定して作成した．gzip の各機能をテストするために作成した入力データの数を表 2 に示す．

表 2: gzip に対して作成した入力セット

gzip の機能	入力データ数
F1	1920
F2	192
F3	24
F4	8
その他	182
合計	2326

各機能 (F1〜F4) に関する入力データの数は，その機能において使用可能なオプションのべき集合の数となる．ファイルの圧縮時には圧縮率を変更することが可能であるため，その入力データの数が他の場合よりも多くなった．gzip へ与えるファイルは，その動作によって変化させた．F1 に関しては，空ファイル，ファイル内が冗長なファイル，圧縮ファイル，通常ファイルを無作為に組み合わせたものを与えた．また，F2〜F4 に関しては，上記のファイルを gzip 形式で圧縮したものを無作為に組み合わせて与えた．そして，その他の入力データとして異常終了を想定して作成したものは，故意に壊した圧縮ファイルを与えるもの，gzip へオプション等の引数を与えないものや存在しないファイルを与えるもの等である．

表 3: 入力セットの命令網羅率

ファイル	命令網羅率 [%]	行数
`bits.c`	100	35 / 35
`deflate.c`	99.3	141 / 142
`gzip.c`	64.5	381 / 591
`inflate.c`	83.5	318 / 381
`lzw.c`	0	0 / 7
`trees.c`	92.0	289 / 314
`unlzh.c`	0	0 / 174
`unlzw.c`	0	0 / 112
`unpack.c`	0	0 / 67
`unzip.c`	35.7	25 / 70
`util.c`	53.6	74 / 138
`zip.c`	100	40 / 40
合計	62.9	1303 / 2071
	76.1	1303 / 1711

「行数」欄は 実行された行数 / 実行可能な行数を示す．

作成した入力セットについての命令網羅率を表 3 に示す．命令網羅率とは，入力セットがプログラムの実行ステートメントをどれだけ網羅したかを示すものである．

表 3 において，`lzw.c`，`unlzh.c`，`unlzw.c`，`unpack.c` の 4 つのソースプログラムの命令網羅率はいずれも 0% であった．これらのファイルの関数は，gzip 形式以外のファイルの圧縮・伸張をするものであり，この評価実験において使用したバージョン 1.2.4 の gzip においては既に使われないものが大部分である．従って，作成した入力セットでこれらのファイル内の関数が実行されることはなかった．また，同様の原因によって，`gzip.c`，`unzip.c`，`util.c` の一部分が実行されることはなく，それらのファイルが高い命令網羅率を示すことはできない．

上記のことを考慮して，`lzw.c`，`unlzh.c`，`unlzw.c`，`unpack.c` の各ソースプログラムを除いた命令網羅率は 76% となる．この命令網羅率は表 3 中の最後の行に示している．`gzip.c`，`unzip.c`，`util.c` の一部分が実行されないことを考慮すると，ここで作成した入力セットは網羅的なテストを行えるといえる．

4.2.2 ミュータントの生成

gzip へミュータントオペレータを適用してミュータントを生成した．ここで適用したミュータントオペレータを以下に示す．これらは，Agrawal によって定義された C 言語用のミュータントオペレータの一部である．

SCRB `continue` の代わりに `break` を間違って使用するエラーのモデル

SBRC `break` の代わりに `continue` を間違って使用するエラーのモデル

Ocor Ocor は二項演算子に対するミュータントオペレータのカテゴリである．この Ocor に属するミュータントオペレータは表 4 内の二項演算子の間違った使用のモデルである．8 個のミュータントオペレータがこの Ocor に属する．例えば，`+` は `-`，`*`，`/`，`%` にそれぞれ置き換えられ，このとき 4 つのミュータントが生成できる．

OIOM インクリメント演算子 “`++`” の誤った使い方によるエラーのモデル

ODOM デクリメント演算子 “`--`” の誤った使い方によるエラーのモデル

OLNG 演算子 `&&`，`||` を用いた算術式において，真偽値の値を逆に間違ったエラーのモデル

OCNG `switch` 文以外の制御構造文分岐の条件式の結果を逆に取り間違えたエラーのモデル

OBNG 演算子 `&` や `|` を使ったビット演算式において，ビットの意味を逆に取り違えたエラーのモデル

VAVR 算術数値変数の誤った使用のモデル

VARR 配列の誤った使用のモデル

VSRR 構造体型の変数の誤った使用のモデル

VPRR ポインタ型の変数の誤った使用のモデル

CCCR 定数の誤った使用のモデル

表 4: 二項演算子に対するミュータントオペレータのカテゴリ Ocor における演算子の分類

分類	演算子
Arithmetic	`+ - * / %`
Bitwise	`| & ^`
Logical	`|| &&`
Shift	`<< >>`
Relational	`< > <= >= == !=`
Arithmetic assingment	`+= -= *= /= %=`
Bitwise assingment	`|= &= ^=`
Plain assingment	`=`
Shift assingment	`<<= >>=`

`lzw.c`，`unlzh.c`，`unlzw.c`，`unpack.c` の各ソースプログラムが実行されることはないので，これらのソースプログラムについてはミューテーションを行わなかった．

今回の評価実験では Agrawal らによって定義された 77 個のミュータントオペレータの内，プログラマが間違い易そうなものを 20 個選択して使用した．使用しなかったミュータントオペレータとしては，ステートメントを 1 行削除するもの，複文の閉じ括弧の位置をずらすもの等がある．

統合テストでは，各モジュールを組み合わせたときに生じる特有のエラーがある．Agrawal らは統合テスト特有のエラーのモデル化を考慮していない．しかし，このようなエラーをモデル化したミュータントを摘出するテストケースは，統合テストにおいて有効であると考えられる．

4.2.3 ミュータントプログラムのテスト

4.2.2 節で示す各ミュータントプログラムを，4.2.1 節で示す各入力データで実行した．そして，これから摘出行列を作成した．入力セットとミュータントの集合から 3.2 節の手順に従い摘出行列を作成した．入力セット内の入力データの数は 2326 個，ミュータントの数は 9305 であり，ここで作成された摘出行列のサイズは 9305 × 2326

表 5: gzip に対し生成したミュータントの数

ミュータントオペレータのカテゴリ	ミュータントの数
ステートメント	41
演算子	3952
変数	4072
定数	1240
合計	9305

であった．

1 つのミュータントプログラムについて 2326 個の入力データを全て実行したときにかかる実行時間は 15 分から 25 分であった．このことから，9305 個の全てのミュータントプログラムについてかかる時間は約 100 日となる．この問題を解決するために，この評価実験では 135 台の計算機を使用した．この環境で，全ての入力データに対して，全てのミュータントプログラムの終了までに約 1 日かかった．

以上から，各入力データがどのミュータントを摘出するかを表した摘出行列を作成した．この摘出行列から，この 2326 個の入力データが摘出したミュータントの数は 5894 個である．従って，この入力セットのミューテーションスコアは 63.4%である．

等価ミュータントについて　gzip に対して生成した全てのミュータントを入力セットが摘出できない，すなわち摘出行列の行ベクトルの要素が全て 0 であるとき，その行ベクトルに対応するミュータントは等価ミュータントであるかもしれない．しかし，そのミュータントが等価ミュータントであるかどうかは人手により判断するしかない．このため，このようなミュータントは摘出することができなかったものとして扱った．

4.2.4　テストセットの構成

摘出行列を用いて，3.3 節で述べた提案手法を用いて，2326 個の入力データへ選別を行った．表 2 に示した元の入力セットへ本手法を適用して選択した入力セットの数を表 6 に示す．

表 6: 本手法を用いて選択した入力データ

gzip の機能	元の入力データ数	選択した入力データの数
F1	1920	8
F2	192	3
F3	24	4
F4	8	2
その他	182	12
合計	2326	29

2326 個の入力データ群から，そのミューテーションスコアと同等である最小の部分集合は 29 個であった，これより，各入力データとそれに対する正解出力を作成することによってテストセットが構成される．

4.3　テストセットの評価

この構成手法によって，2326 個の入力データから 29 個の入力データを選択した．この 29 個が初期入力セットのミューテーションスコアを維持する最小の部分集合である．

初期入力セットと本手法で構成した入力セットの比較を表 7 に示す．命令網羅率に関しては，本手法で構成した入力セットと初期入力セットで値がほぼ等しいことが分かる．従って，本手法で構成した入力セットで初期入力セットが実行できるステートメントはほぼ変化しない．また，この 29 個の入力データは人手で 75%の命令網羅率を示す入力セット作成するときよりも少ない数であると考えられる．

表 7: 初期入力セットと本手法で構成した入力セットの比較

	初期入力セット	選択された入力セット
入力データの数	2326 個	29 個
ミューテーションスコア	63.4%	63.4%
命令網羅率	76%	75%

互いの正解出力作成コストを考えると，本手法で構成した入力セットの正解出力作成コストは大きく減少すると言える．そして，29 個の入力データをオリジナルプログラムで実行するときにかかる時間は 20 秒であった．これに対し，元の 2326 個の入力データをオリジナルプログラムで実行するときにかかる時間は約 15 分であり，テスト時間でも約 20%減少した．以上から，本提案手法によりテストセットを構成することで，正解出力作成コスト・テスト実行コスト・バグの検出能力の 3 つの側面から効果的なテストが行える．

5　おわりに

5.1　まとめ

テストセットは，複数のテストケースを組み合わせることによって構成される．テストセット内のテストケースの数が多ければ，様々なバグを検出することが期待できる．しかし，入力データに対する正解出力の作成にかかるコストとテスト時間を考えたとき，テストケースの数を単純に増やすことは推奨されない．

本稿ではテストセットのバグ検出能力をミューテーションスコアとして捉える．そして，与えられた入力セットからミューテーションスコアを下げない最小の部分集合を求めることで，テストセットの構成を支援する方法を提案する．これにより，元のバグ検出能力を維持したまま，テストケースの数を抑えることが可能である．

そして，gzip の入力データ群を作成し，本手法を用いた評価実験を行った．ここでは，2326 個の入力データを作成したが，本手法を用いることにより 29 個に減少させた．この各入力データに対する正解出力を作成して，これをテストセットとして用いれば，効率的なテストの実施が期待される．

5.2　今後の課題

gzip の入力データを作成して本構成法を適用することにより，その正当性を示した．しかし，本構成手法の汎用性を示すためにも，ほかのプログラムの入力セットに本手法を適用することが今後の課題である．

また，ミューテーションスコアはテスト対象プログラムと入力データの 2 つから算出することができるが，どのようなミュータントオペレータを適用するかに依存する．ミュータントオペレータはプログラム作成者，または，プロジェクトごとにより，間違いやすいエラーのモデルになっているものが望ましい．従って，バグレポート等を用いてそのエラーをモデル化したミュータントオペレータを適用することにより，現実的なバグを検出するテストセットの構成に繋がる．

謝辞

ご指導，ご議論頂いた愛知県立大学情報科学部 稲垣康善教授，名古屋大学大学院情報科学研究科 阿草清滋教授，および山本研究室の皆様に感謝します．本研究は、文部科学省リーディングプロジェクト「e-Society 基盤ソフトウェアの総合開発」の支援により行われた。

参考文献

[1] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton and F. Sayward, “Mutation Analysis,” Georgia Tech Report GIT/ICS-79-08, September 1979.

[2] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, “Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs,” Proceedings of the Seventh Conference on Principles of Programming Languages, pp. 220–233, January 1980.

[3] A.J. Offutt and K.N. King, “A Fortran Language System for Mutation-Based Software Testing,” Software Practice and Experience, 21(7):686–718, July 1991.

[4] H.Agrawal R.A. DeMillo B.Hathaway W.Hsu W.Hsu E.W. Krauser R.J. Martin A.P. Mathur and E.Spafford, “Design of Mutant Operator for the C Programming Language,” Technical Report SERC-TR-41-P, Purdue University, West Lafayette, IN, 1989