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

テストセットの品質の尺度を測定する方法として，ミューテーション法が提案されている．本稿ではミューテーション法を用いることにより，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，プログラムへ与えた多数の入力データからバグ検出能力を維持した最小部分集合をテストセットとして用いることを提案する．

A Investigation about Test Set Constituent Support Based on Mutation Testing

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

Graduate School of Information Science and Technology, Aichi Prefectural University

Abstract

As a method to measure a standard of quality of a test set, mutation testing is suggested. We catch bug detective ability of a test set as mutation score with this report by using mutation testing. And We propose that We use the the smallest subset that maintained bug detective ability from a lot of input data which We gave for a program as a test set.

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．そのため，バグの検出時期が遅れるほど，その修正コストは増加する．また，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なん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，単にテストケースを数多くすればいいというわけではない．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．モジュール単位で行われる単体テストでは，プログラム全体のテストである統合テストに比べ正解出力の作成が比較的容易

である．統合テストで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大変な労力を要する．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現在の課題である．

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．逆に，ミュータントを検出できないテストセットのバグ検出能力は低いものとみなせる．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を行うことが期待できる．

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**，ミューテーションされたプログラムを**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．

入力データにおいて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 や 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．

このことに注目すると，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個々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．

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の摘出を調べる．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，その入力データに対する正解出力からなるテストケースは，なんらかのバグ検出能力を持っているといえる．

摘出行列とは，ミュータント生成システムで生成された 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 $\boldsymbol{K}$ 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．

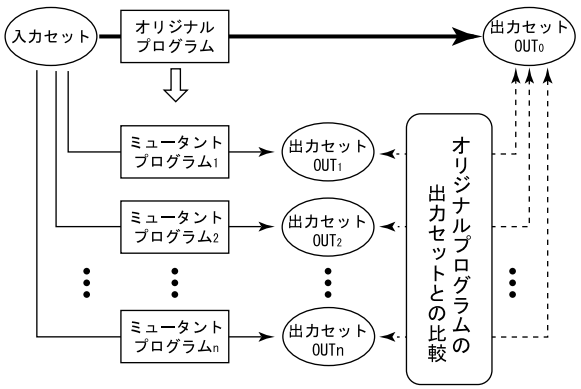

図 1: 摘出行列の作成

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 $(1 \leq j \leq N)$ に対する出力データを out_j $(1 \leq j \leq N)$ 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 上で

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．

3.3 テストセット構成手法

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手順を述べる．ここで，この最小の部分集合のことを入力セットと区別するためにテストデータ群と呼ぶ．初期のテストデータ群の要素は空であり，入力セット内の入力データが各手順において追加される．摘出行列中の各列ベクトルは，各入力データが摘出したミュータントを表している．各入力データ in_j $(1 \leq j \leq N)$ に対応する列ベクトルを $\boldsymbol{In_j}$ とする．そして，各ミュータント m_i $(1 \leq i \leq M)$ に対応する行ベクトルを $\boldsymbol{m_i}$ とする．以下の手順では，各行ベクトルの値が全て 1，または，0 であるものを無視して考える．各行ベクトルの値が全て 1 である場合は，全ての入力データでそのミュータントを摘出できることを意味している．また，全ての値が 0 であるものは，そのミュータントを摘出する入力データが不足しているか，そのミュータントが等価ミュータントであることを示している．両方の場合について，最小部分集合を求めるときは，これらを無視できる．

ステップ 1: 入力データに対応する列ベクトルについて，ある列ベクトル $\boldsymbol{In_j}$ が別の列ベクトル $\boldsymbol{In_k}$ $(j \neq k)$ に包含されているとき，すなわち，それらのベクトルの各要素 In_{j_i}，In_{k_i} が以下の式を満たすとき，列 j を削除する．

$$\forall i \quad In_{j_i} \leq In_{k_i} \quad (1 \leq i \leq M)$$

$\boldsymbol{In_j}$ が $\boldsymbol{In_k}$ に包含されているということは，$\boldsymbol{In_j}$ に対応する入力データ in_j が摘出できる全てのミュータントを $\boldsymbol{In_k}$ に対応する入力データ in_k でも摘出できることを示している．従って，入力データ in_j はテストデータ群には不要である．

ステップ 2: 1 つの要素のみが 1 であるような行ベクトル $\boldsymbol{m_i}$ を探す．

$\boldsymbol{m_i}$ がある　→　**ステップ 3** へ
$\boldsymbol{m_i}$ がない　→　**ステップ 5** へ

これは，ミュータント m_i を摘出する入力データが唯一であり，その入力データは最小部分集合に必須な要素であることを示す．

ステップ 3: 行ベクトル $\boldsymbol{m_i}$ に対応するミュータント m_i を唯一摘出することができる入力データ in_j をテストデータとしてテストデータ群に追加する．そして，**ステップ 2** に戻る．

このとき，この入力データ in_j に対応する列 j を摘出行列から削除する．そして，in_j が摘出する全てのミュータントに対応する行を摘出行列から削除する．これは，入力データ in_j がそれらのミュータントも摘出するので，残りの入力データがそれらを摘出する必要はないことに対応する．

ステップ 4: 要素の和が最大となる列ベクトル $\boldsymbol{In_j}$ を探し，$\boldsymbol{In_j}$ に対応する入力データ in_j をテストデータ群に追加する．これは，入力データの中で最も多くのミュータントを摘出する入力データを最小部分集合に採用することに対応する．

このとき，列 j を摘出行列から削除する．そして，in_j が摘出する全てのミュータントに対応する行ベクトルを摘出行列から削除する．これは，ステップ 3 と同じ処理である．

ステップ 5: 摘出行列が空であるかどうかを調べる．

摘出行列が空　→　終了
摘出行列が空ではない　→　**ステップ 4** へ戻る．

終了時のテストデータ群，元の入力セットのミューテーションスコアを下げない最小の部分集合である．

4 評価実験

4 章で提案したテストデータ構成法を用いて，実際に gzip-1.2.4 (以下 gzip) のテストデータを構成する．そして，構成されたテストデータ群についてこの構成手法を評価する．

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．圧縮時に Lempel-Ziv コーディング (LZ77)*を利用する．

gzip が提供する機能は，次の 4 種に分類できる．

F1: ファイルを圧縮

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

gzip の終了状態は正常に終了した場合は 0，エラーが起きた場合の終了状態は 1 である．また，警告が起きた場合は，その終了状態は 2 となる．

4.2 実験の手順

本研究で提案するテストデータ群構成法を用いて 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. テストデータ群の構成.

評価実験は，表 1 の環境の元で行った．

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 の各機能をテストするために作成した．また，それ以外にも，ヘルプの表示や以上終了の場面を想定して作成ものもある．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．

*Lempel-Ziv コーディング (LZ77) は 1977 年に Abraham Lempel 氏と Jacob Ziv 氏が発表したデータ圧縮アルゴリズムである．

表 1: 評価実験の環境

Sun Blade 2000

CPU	UltraSPARCⅢCu
メモリ	1.0GB

表 2: gzip に対して作成した入力セット

F1	1920
F2	192
F3	24
F4	8
その他	182
total	2306

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パス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ほとんどである．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，それらの命令網羅率が低くなっている．

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．`gzip.c`，`unzip.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が網羅的なテストを行えるといえる．

表 3: 入力セットの命令網羅率

ファイル	命令網羅率 [%]	ライン
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
total	62.9	1303 / 2071

4.2.2 ミュータントの生成

gzip からミュータントを生成した．ミュータントを生成するために，CASE ツールプラットフォームである Sapid を利用してソースプログラムの解析をした．そして解析結果を基にミュータントオペレータに従った対象要素へミューテーションを行った．ここで適用したミュータントオペレータは，Agrawal によって定義される C 言語用のミュータントオペレータの一部である．また，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．

表 4: 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

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 であった．

摘出行列を作成することは，各入力データが各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．ループの終了条件となる要素へのミューテーションにより，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プログラムも摘出されたものとして扱った．従って，この評価実験において各入力データがミュータントを摘出することは，次の 2 つの条件のどちらかを満たしたときである．

- 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．

  この評価実験で設けた時間は，各入力データでのオリジナルプログラムにおける最大の実行時間の 2 倍とした．

  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．

一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．この摘出行列のサイズは 31M であった．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　テストデータ群の構成

摘出行列を用いて，**??**節で述べた提案手法を用いて，2326 個の入力データからテストデータを構成する．

ステップ 1 では各列の包含関係を調べることにより，他の列によって包含される列を摘出行列から削除する．しかし，ここで削除される列はなく，互いの列が包含関係を持っていなかった．

ステップ 2 とステップ 3 において，10 個の入力データがテストデータ群に追加される．この時に摘出行列の行と列が削除されるが，元の 31M の摘出行列が 4.1M に縮小される．

そして，ステップ 4 とステップ 5 でテストデータ群に追加される入力データの数は 19 個であった．

以上より，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 個であり，この 29 個が本手法で構成したテストデータ群である．

4.3　テストデータの評価

この構成手法によって，2326 個の入力データから 29 個の入力データをテストデータとして選択した．ミューテーションスコアに注目すると元の入力セットと等しく 63.4%であった．また，本手法によって選択した 29 個の入力データの命令網羅率を測定した．これを表 5 に示す．この表 5 では，ミューテーションを行わなかった `lzw.c`，`unlzh.c`，`unlzw.c`，`unpack.c` については記述していない．元の入力セットの命令網羅率である表 3 と表 5 を比較したとき，`gzip.c` の命令網羅率が若干下がるものである．しかし，全体の命令網羅率はほぼ同じであるといえる．

表 5: 本手法により選択された入力データの命令網羅率

ファイル	命令網羅率 [%]	ライン
`bits.c`	100	35 / 35
`deflate.c`	99.3	141 / 142
`gzip.c`	62.9	372 / 591
`inflate.c`	83.5	318 / 381
`trees.c`	92.0	289 / 314
`unzip.c`	35.7	25 / 70
`util.c`	53.6	74 / 138
`zip.c`	100	40 / 40
total	75.6	1294 / 1711

元の入力セットのミューテーションスコアと，本手法により選択したテストデータ群のミューテーションスコアは等しいものである．そして，この 29 個がこのミューテーションスコアを保持するための最小の集合である．単純に入力データの数に注目したとき，その数は約 1%になった．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．

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データの数が多ければ，様々なバ

グを検出することが期待できる．しかし，テスト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，テストデータの数を単純に増やすことは推奨されない．そこで，テストセットのバグ検出能力とテストセット内のテストデータの数を考慮したテストセットの構成法を提案した．

本研究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．与えられた入力セット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求めることにより，元のバグ検出能力を維持したまま，テストデータ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．この各テスト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テストデータの 2 つから算出することができるが，さらには，どのようなミュータントオペレータを与えるかによってもその数値に影響を与える．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化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によって，より現実的な間違いをモデルがすることができる．これは，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

御指導頂いた，愛知県立大学情報科学部 稲垣康善教授と名古屋大学院情報科学研究科 阿草清滋教授に感謝します．

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