情報処理学会論文誌　Vol.61　No.10　1617–1628 (Oct. 2020)

命令削除ミューテーションに基づく欠陥局所化の産業用ソフトウェアにおける評価

徳本 晋[1,2,3,a)]　本位田 真一[2,3,b)]

受付日 2020年2月12日, 採録日 2020年7月7日

概要： 欠陥局所化技術はテスト結果やテスト実行情報などから欠陥の原因個所の候補を順位付けすることでデバッグの作業コストを削減するための技術である．いくつかの欠陥局所化技術のなかで，ミューテーション解析に基づく欠陥局所化技術（MBFL）は高い精度で欠陥を局所化できるが，実行コストが高い問題がある．一方，ミューテーション解析において，命令削除ミューテーションオペレータはミューテーション箇所の偏りが少なく，単独での利用でもすべてのミューテーションオペレータを使った場合と同等の効果があることが知られている．本論文では命令削除ミューテーションオペレータのみを用いたMBFL（SDL-MBFL）を提案し，実際の製品で使われているソフトウェアと実際に起こった9件の欠陥に対してSDL-MBFLによる局所化の評価を行った．評価の結果として，SDL-MBFLは欠陥箇所の候補の上位100位という基準において既存手法より多くの欠陥箇所をあげられた．

キーワード： ソフトウェアテスト，ミューテーションテスト，欠陥局所化

Evaluating Statement Deletion Mutation-based Fault Localization in Industrial Software

Susumu Tokumoto[1,2,3,a)]　Shinichi Honiden[2,3,b)]

Received: February 12, 2020, Accepted: July 7, 2020

Abstract: Fault localization is a technique to reduce the work cost of debugging by ranking the candidates of the cause of the defect based on the test result and the test execution information. Among several fault localization techniques, mutation based fault localization (MBFL) can localize defects with high accuracy, while mutation analysis has a problem of high execution cost. In the context of mutation analysis, the statement deletion mutation operator has less deviation of the mutation position and the same effect as the use of all mutation operators. In this paper, we propose statement deletion mutation based fault localization, SDL-MBFL, and evaluate it on industrial software used in actual products and nine defects actually occurred. As a result of the evaluation, the SDL-MBFL can find more defects than the existing methods in high ranking.

[1] 株式会社富士通研究所
FUJITSU LABORATORIES LTD., Kawasaki, Kanagawa 211–8588, Japan
[2] 早稲田大学
Waseda University, Shinjuku, Tokyo 162–0042, Japan
[3] 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[a)] tokumoto.susumu@fujitsu.com
[b)] honiden@nii.ac.jp

1. はじめに

ソフトウェアテストはプログラムを実行することで誤った振舞いを見つけるプロセスである．テストの最小の実行単位はテストケースと呼ばれる入力，実行条件，実行手続き，期待結果の組とする．テストケースの集合（テストスイート）を事前に用意し，そのテストケースの入力，実行条件，実行手続きに従いプログラムを実行することで得た出力がテストケースの期待結果と一致するかを確認する．プ

© 2020 Information Processing Society of Japan

ログラムの出力が期待結果と一致するテストケースを成功テストケース，一致しないテストケースを失敗テストケースと呼ぶ．またプログラムの出力と期待結果が一致しない原因のことを欠陥と呼ぶ．テストでは欠陥の存在を見つけることはできるが，その欠陥がどこにあるかを特定することはできない．

ソフトウェアのデバッグ作業のうち，欠陥の原因を特定することは高度な知識が要求され時間もかかる困難な作業である．特に産業界においては，同じ人間が継続して保守するとは限らず，時にはプログラムの構造を把握する者がなくブラックボックスとして扱われるようなソフトウェアに対し，欠陥の原因を探り当てないとならない場合もある．

欠陥局所化技術は欠陥の原因箇所の候補を自動で特定することで，デバッグにおける人の作業コストを削減する技術である．代表的な手法として，スペクトルベース欠陥局所化（Spectrum-based Fault Localization：SBFL）[1] とミューテーションベース欠陥局所化（Mutation-based Fault Localization：MBFL）[2], [3] が存在する．スペクトルベース欠陥局所化の基本的なアイデアは，各テストケースの実行トレースと実行結果を記録し，失敗テストケースが多く通る命令をより怪しいと見なし，成功テストケースが多く通る命令はより安全だと見なす．ミューテーションベース欠陥局所化は，スペクトルベース欠陥局所化にミューテーション解析を要素技術として取り入れ，拡張したものである．ミューテーション解析は，ミュータントと呼ばれる人為的に欠陥を注入したプログラムに対しテストし，その欠陥をテストで検出できるかを調べることでテストケースの欠陥検出能力を評価することを目的とするが，欠陥局所化で用いる場合はどのミューテーション箇所がどのテストケースの結果に影響を与えるかを記録し，失敗テストケースへの影響が大きいミューテーション箇所はより怪しいと見なし，成功テストケースが多く通るミューテーション箇所はより安全だと見なすことで，各命令の怪しさを順位付けする．

しかしミューテーション解析およびそれを用いるミューテーションベース欠陥局所化はミュータントの生成とテストを繰り返すため，実行コストが高いという課題がある．ミューテーション解析の実行コストを減らすために，注入する欠陥の種類（ミューテーションオペレータ）を命令削除ミューテーションオペレータのみに限定しミュータントを減らす方法が知られている [4], [5].

本論文では命令削除ミューテーションオペレータをミューテーションベース欠陥局所化に用いる手法（SDL-MBFL）を提案する．評価においては，企業におけるシステム再構築プロジェクトでの実際のソースコード，テストケース，欠陥を用いる．既存の欠陥局所化の研究における評価ではOSS を用いるものがほとんどであり，産業用ソフトウェアを用いた評価は非常に少なく，特にミューテーションベース欠陥局所化における評価が公開されるのは我々の知る限り存在しない．対象プロジェクトの特徴として，複数の欠陥を同時に特定する必要がある．これは，ここで「欠陥」と呼んでいるのは再構築前後の非互換のことを意味しており，本来直すべきものではない非互換も含まれているため，1 つずつ特定しては修正するということができないからである．評価の結果として，SDL-MBFL は MBFL に比べ 20.3%の実行時間削減が可能であり，また，他の欠陥局所化手法よりも怪しさの順位が 100 位以内における欠陥箇所の特定は優れていた．これは複数の欠陥に対する SDL-MBFL が，実行コストを抑えつつ優れた欠陥局所化性能を持つ可能性を示唆している．

まとめると，本論文の貢献は以下になる．

- 命令削除ミューテーションを用いたミューテーションベース欠陥局所化の提案と実装を行った
- 企業において実際の製品となっているソフトウェアと複数の欠陥を用いて，SDL-MBFL を含む欠陥局所化技術を比較した
- 上記比較にて，SDL-MBFL は従来の MBFL と比べ同等の欠陥検出数で，実行時間を 20.3%削減できた

以降の本論文の構成について述べる．2 章で本研究の背景として命令削除ミューテーションに基づく欠陥局所化について紹介する．3 章ではリサーチクエスチョンと評価の方法について説明し，4 章では評価の結果を示す．5 章では考察を述べ，6 章では関連研究を紹介する．

2. 命令削除ミューテーションに基づく欠陥局所化

本章では，ミューテーション解析と一般的な欠陥局所化について紹介し，命令削除ミューテーションに基づく欠陥局所化について説明する．また，複数の欠陥局所化を組み合わせたハイブリッドな手法についても紹介する．

2.1 ミューテーション解析

ミューテーション解析は，ミュータントと呼ばれる人為的に欠陥を入れたプログラムに対し，テストによってその欠陥を検出できるかを調べることにより，テストの欠陥検出力を測定する．人為的に挿入する欠陥はプログラム言語ごとにミューテーションオペレータと呼ばれるプログラムの変化方法がルールとして定義されている．たとえば，`a + b` というプログラム要素は，OAAN というミューテーションオペレータによって，`a - b` というミュータントを生成する．

Offutt ら [6] はミューテーション解析の高速化の戦略を *do fewer*，*do smarter*，*do faster* の 3 種類に分類した．“do fewer” はミュータントを減らす戦略，“do smarter” は分散させて計算させるなどの戦略，“do faster” はミュータントの生成・テスト実行を早くする戦略になる．“do fewer” の

© 2020 Information Processing Society of Japan

アプローチの代表的なものは，ミューテーションオペレータの選択である．

命令削除ミューテーションオペレータ（SDL）はミューテーションオペレータの 1 つであり，プログラムに命令（Statement）を削除する変更を与える．**図 1** に示す例では，テスト対象プログラムに対し，3 つの命令削除ミューテーションを適用しており，それぞれ 2〜4 行目の if 文全体，3 行目の return 文，5 行目の return 文が削除されたミュータントが生成されている．Deng ら [4] や Delamaro ら [5] によって SDL のみですべてのオペレータを使った場合と同様の効果があることが実験的に示されている．

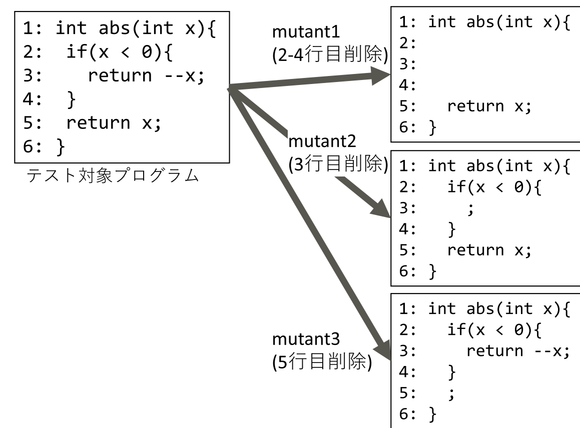


図 1 命令削除ミューテーション
Fig. 1 Statement deletion mutation.

2.2 欠陥局所化

本節では代表的な欠陥局所化技術であるスペクトルベース欠陥局所化とミューテーションベース欠陥局所化とそれらを組み合わせたハイブリッドな手法について説明する．

2.2.1 スペクトルベース欠陥局所化

スペクトルベース欠陥局所化（SBFL）は，テストにおける命令網羅情報（実行可能命令スペクトル）とテストの成否を統計的に扱うことで，欠陥の原因となる箇所の怪しさ（疑惑値：suspiciousness）を測定する技術である．プログラム中の各命令に対し，通過する成功・失敗テストケース数をカウントし，通過する失敗テストケースが多く，通過する成功テストケースが少ないほど高い疑惑値とする．逆に通過する失敗テストケースが少なく，通過する成功テストケースが多いものは低い疑惑値とする．ある命令 s における疑惑値 $Susp(s)$ を計算する式は様々なものが提案されており，以下は代表的なものになる．

$$Tarantula : Susp(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{failed(s)}{totalfailed} + \frac{passed(s)}{totalpassed}} \quad (1)$$

$$Ochiai : Susp(s) = \frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s) + passed(s))}} \quad (2)$$

$$Op2 : Susp(s) = failed(s) - \frac{passed(s)}{totalpassed + 1} \quad (3)$$

$$DStar : Susp(s) = \frac{failed(s)^{*}}{passed(s) + (totalfailed - failed(s))} \quad (4)$$

ここで $passed(s)$ は命令 s を通った成功テストケース数，$failed(s)$ は命令 s を通った失敗テストケース数，$totalpassed$ はすべての成功テストケース数，$totalfailed$ はすべての失敗テストケース数になる．また DStar の $*$ は正の実数を表す変数であり，本論文では 2 とする．

2.2.2 ミューテーションベース欠陥局所化

ミューテーションベース欠陥局所化（MBFL）は，SBFL と同じようにテストの成否を用いて欠陥の原因箇所を求めるが，その際にトレース情報だけではなくミューテーション解析の結果も用いる．通常のミューテーション解析では，元のプログラムで失敗するテストケースは存在しないものとして，ミュータントと呼ばれる人為的に欠陥を入れたプログラムに対しテストスイートを実行したときに失敗するか，つまり，その欠陥を検出できるかを調べることにより，テストスイートの欠陥検出力を測定するが，MBFL では元のプログラムで失敗するテストケースを含めてミュータントを実行し，その失敗テストケースの出力に与える影響が大きいミュータントほど，そのミューテーション箇所における疑惑値は大きくなるような計算をし，欠陥箇所の順位付けを行う．よって，通常のミューテーション解析のようにミュータントを殺せるテストケースが 1 つでも見つかればよいわけではなく，すべてのテストケースにおいてミュータントが殺せたかの情報が必要となる．また，ミュータントは 1 命令ごとに複数存在する場合がある．

MBFL の代表的な手法として MUSE [2] と Metallaxis [3] がある．

MUSE は以下の式でミュータント m_i における疑惑値を算出する．

$$Susp(m_i) = failed(m_i) - \frac{f2p}{p2f} \cdot passed(m_i) \quad (5)$$

ここで $failed(m_i)$ は元のプログラムで失敗していたテストケースのうちミューテーション m_i によって成功に変化したテストケース数，$passed(m_i)$ はその逆で，元のプログラムで成功していたテストケースのうちミューテーション m_i によって失敗に変化したテストケース数である．$f2p$ は元のプログラムで失敗していたテストケースのうち，任意のミューテーションによって成功に変化したテストケース数，$p2f$ はその逆で，元のプログラムで成功していたテストケースのうち，任意のミューテーションによって失敗に変化したテストケース数である．ミュータントごとの疑惑値を，各命令ごとへの疑惑値へ変える場合は，$Susp(s) = Avg_{m_i \in mut(s)} Susp(m_i)$ とする．ここで $mut(s)$ は命令 s におけるミュータントの集合とする．

© 2020 Information Processing Society of Japan

一方，Metallaxis は Ochiai の式をベースに以下の式でミュータント m_i における疑惑値を算出する．

$$Susp(m_i) = \frac{failed(m_i)}{\sqrt{totalfailed \cdot (failed(m_i) + passed(m_i))}} \tag{6}$$

本論文では，Ochiai をベースとした式 6 のほかに，Tarantula，Op2，DStar をベースにした以下の式についても用いる．

$$Tarantula : Susp(m_i) = \frac{\frac{failed(m_i)}{totalfailed}}{\frac{failed(m_i)}{totalfailed} + \frac{passed(m_i)}{totalpassed}} \tag{7}$$

$$Op2 : Susp(m_i) = failed(m_i) - \frac{passed(m_i)}{totalpassed + 1} \tag{8}$$

$$DStar : Susp(m_i) = \frac{failed(m_i)^*}{passed(m_i) + (totalfailed - failed(m_i))} \tag{9}$$

ここで $failed(m_i)$ は元のプログラムで失敗していたテストケースのうち，ミューテーション m_i によって元のプログラムの出力と異なる出力となるテストケース数，$passed(m_i)$ は元のプログラムで成功していたテストケースのうち，ミューテーション m_i によって元のプログラムの出力と異なる出力となるテストケース数，$totalfailed$ は元のプログラムでのすべての失敗テストケース数，$totalpassed$ は元のプログラムでのすべての成功テストケース数である．

MUSE と Metallaxis の違いは，ミューテーションによる出力への影響をどこまで確認するか，の違いともいえる．MUSE は失敗していたテストケースを成功にさせるミューテーション，つまり出力を期待結果に一致させるミューテーションの箇所の疑惑値を高くする．一方，Metallaxis は失敗していたテストケースの出力がミューテーションにより変化さえしていれば，期待結果に一致していなくても，そのミューテーションの疑惑値を高くする．そのような違いから，両者を比較したときの一般的な傾向としては，MUSE は偽陰性が多く，逆に Metallaxis は偽陽性が多い．文献 [3], [7] の Metallaxis と MUSE の比較実験において，Metallaxis の方が優れた結果となっているため，本論文では Metallaxis を対象として扱う．

2.2.3 命令削除ミューテーションベース欠陥局所化

命令削除ミューテーションベース欠陥局所化（SDL-MBFL）は，ミューテーションベース欠陥局所化において，命令削除のミューテーションオペレータ（SDL）のみを用いる方法である．通常の MBFL では，1 命令に複数のミューテーションがあり得ることや，オペレータの選び方によっては逆にまったくミューテーションが行われない命令が存在することがあり，実行コストや欠陥局所化性能に影響を及ぼしていた．そこで，SDL を用いることで，1 命令に必ず 1 つのミューテーションを行うような MBFL を実現した．

	テスト対象プログラム	テストケース(TC)と各TCで実行した命令						成功TC数	失敗TC数	SBFL (Tarantula) 疑惑値	順位
	int mid(int x, int y, int z){	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3				
S_1	int m;	●	●	●	●	●	●	4	2	0.500	7
S_2	m = x; //m = z;	●	●	●	●	●	●	4	2	0.500	7
S_3	if (y < z)	●	●	●	●	●	●	4	2	0.500	7
S_4	if (x < y)	●	●			●	●	2	2	0.667	3
S_5	m = y;		●					1	0	0.000	13
S_6	else if (x < z)	●				●	●	1	2	0.800	1
S_7	m = y; //m = x;	●					●	1	1	0.667	3
S_8	else			●	●			2	0	0.000	13
S_9	if (x > y)			●	●			2	0	0.000	13
S_{10}	m = y;			●				1	0	0.000	13
S_{11}	else if (x > z)				●			1	0	0.000	13
S_{12}	m = x;							0	0	0.000	13
S_{13}	return m;	●	●	●	●	●	●	4	2	0.500	7
	}	P	P	P	P	F	F				

図 2 スペクトルベース欠陥局所化

Fig. 2 Spectrum based fault localization.

	テスト対象プログラム	テストケース(TC)と出力が変化したミュータント						削除する命令	成功TC数	失敗TC数	SDL-MBFL 疑惑値	順位
	int mid(int x, int y, int z){	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3					
S_1	int m;							–	0	0	0.000	13
S_2	m = x; //m = z;				✔	✔		S_2	1	1	0.667	4
S_3	if (y < z)		✔	✔			✔	S_3-S_{12}	2	1	0.500	6
S_4	if (x < y)		✔				✔	S_4-S_7	1	1	0.667	4
S_5	m = y;		✔					S_5	1	0	0.000	13
S_6	else if (x < z)						✔	S_6-S_7	0	1	1.000	2
S_7	m = y; //m = x;						✔	S_7	0	1	1.000	2
S_8	else			✔				S_8-S_{12}	0	0	0.000	13
S_9	if (x > y)			✔				S_9-S_{12}	1	0	0.000	13
S_{10}	m = y;			✔				S_{10}	1	0	0.000	13
S_{11}	else if (x > z)							S_{11}-S_{12}	0	0	0.000	13
S_{12}	m = x;							S_{12}	0	0	0.000	13
S_{13}	return m;	✔	✔	✔	✔	✔	✔	S_{13}	4	2	0.500	6
	}	P	P	P	P	F	F					

図 3 命令削除ミューテーションベース欠陥局所化

Fig. 3 Statement deletion mutation based fault localization.

図 2 は SBML での欠陥局所化の例，**図 3** は SDL-MBFL での欠陥局所化の例で，共通で対象となる関数 `mid()` は 3 つの引数 `x`，`y`，`z` のうち，中央値となる値を返すことを期待結果とする．欠陥は S_2 と S_7 に存在し，それぞれ `m = z;` と `m = x;` となっていたものが誤って入れ替わってしまっている．これにより 5 つのテストケースのうち，`(5,3,4)` と `(2,1,3)` のテストケースが失敗となっている．図 2 において各テストケースで実行された命令を ● で表している．図 3 では，各命令に対応する命令削除ミュータントが存在し，√ は元のプログラムの出力と異なる出力に変化させるミュータントを表している．

SDL-MBFL は Metallaxis に Tarantula の疑惑値の算出式を適用したもので計算している．例のなかの S_2 を削除

© 2020 Information Processing Society of Japan

したもの，S_7 を削除したものが，それぞれ疑惑値の順位が4位，2位となっており，SBFL における順位より高いことが分かる．これは，SDL によって網羅的に各命令に対しミューテーションが行われており，かつ，各命令が変化したときの影響を測定することで，成功テストケースが多く通るような命令に対しても局所化が成功できている．また，各命令で複数のミューテーションが行われてないことから効率性に関しても優れていることも分かる．

さらに，SDL-MBFL だけではなく，MBFL 一般の利点の1つとして，各ミュータントに対し独立して出力へ与える影響を疑惑値として算出するため，複数の欠陥においてもそれぞれ独立して影響を測定でき，SBFL よりも優れた欠陥局所化が可能であると考えられている [2].

2.2.4 MBFL と SBFL のハイブリッド手法

Pearson ら [7] によって，MBFL と SBFL を組み合わせたハイブリッドの手法として以下が提案された．

- Hybrid-Failover：ミューテーションができない命令の疑惑値は SBFL の値を使う．
- Hybrid-Average：MBFL と SBFL のそれぞれの疑惑値の平均をとる．
- Hybrid-Max：MBFL と SBFL の疑惑値を比較し，大きい方を採用する．

Pearson らは，このうち Hybrid-Average が最も優れているという実験結果を得ている．

3. 評価手法

本章では欠陥局所化の評価の方法について説明する．

3.1 リサーチクエスチョン

評価にあたり，以下のリサーチクエスチョンに対して調査を行った．

- RQ1：各ミューテーション解析の実行時間はどのくらいか？
- RQ2：SDL-MBFL の疑惑値の算出式はどれが良いか？
- RQ3：SDL-MBFL は他の欠陥局所化手法と比較して欠陥箇所を上位にあげられるか？
- RQ4：SDL-MBFL と SBFL のハイブリッド手法は欠陥箇所を上位にあげられるか？

RQ1 は，SDL-MBFL の実行速度が他の MBFL の実行速度と比較してどの程度速くなるかを調査する．RQ2 は，SDL-MBFL を使うときに，式 (6)～(9) で示した疑惑値の算出式の中で実際の欠陥を高い順位にあげられるのはどれかを調査する．RQ3 は，比較対象として SBFL，MBFL，SDL なしの MBFL の欠陥局所化性能を測定し，SDL-MBFL と比較して実際の欠陥を高い順位にあげられるかを調査する．RQ4 は，SDL-MBFL と SBFL のハイブリッド手法の欠陥局所化性能を測定し，どのような組合せ方が良いかを調査する．

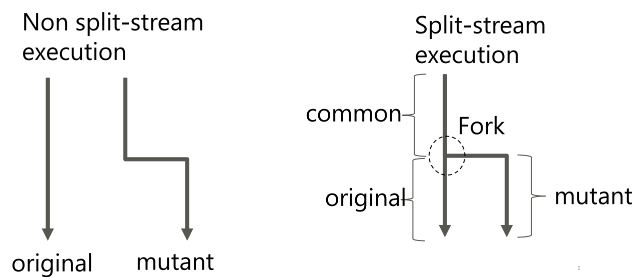


図 4 ストリーム分割実行
Fig. 4 Split-stream execution.

3.2 ツール

Tokumoto らが開発した MuVM [8] をベースとして，C 言語のプログラムのためのミューテーションベース欠陥局所化のツールを実装した．MuVM は以下の特徴により高速なミューテーション解析を実現するツールである．

- ミューテーション用仮想機械
- ストリーム分割実行
- 実行時ミューテーション適用
- メタミューテーション

ストリーム分割実行は，ミューテーション用仮装機械（VM）上で実行状態を保存しながら実行し，各命令の実行時にミューテーションした命令を実行する状態へ分岐する手法である．図 4 のように，元のプログラムの実行とミュータントの実行を分岐まで共通化することでテスト実行時間の短縮が可能になる．また実行時ミューテーション適用は，ソースコードを書き換えず，中間表現の命令実行時に各実行状態のミュータント情報から命令を読み替えて実行する手法である．これにより，通常ミュータントの個数分のコンパイルが必要であるのを，1回のコンパイルで十分となるためコンパイル時間の短縮が可能になる．さらに，実行時ミューテーション適用における課題として，ソースコード上の情報がコンパイル時の最適化などによって中間表現では失われ，ソースコード上では可能なミューテーションが中間表現では適切に行われないことがあるが，MuVM ではコンパイル前にメタミューテーション関数を埋め込むことで最適化による消失を避け，ソースコードへのミューテーションと一致する中間表現レベルでのミューテーションを可能にした．

3.3 評価対象

評価の対象となるシステム再構築プロジェクトの概要を図 5 に示す．このシステム再構築プロジェクトでは，元々サーバ製品とストレージ製品で別々に使われていた SMTP ライブラリを新たに作り直し共通化する．この対象は文献 [9] で用いた評価対象と同様のものである．

システム再構築による非互換個所の確認のため，移行前のサーバ向けのプログラムに対してシンボリック実行ツール KLEE [10] を適用し，生成したテストケースを移行後の

© 2020 Information Processing Society of Japan

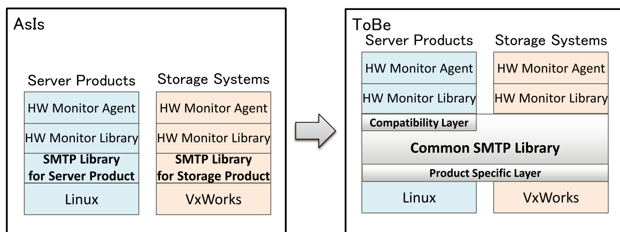


図 5　システム再構築
Fig. 5　Overview of system re-engineering project.

表 1　対象プログラムの概要
Table 1　Overview of subject program.

用途	サーバ監視装置向け SMTP ライブラリ
言語	C 言語
規模	13KLOC
実行可能行数	5,496 LOC
全テストケース数	10,876 件
失敗テストケース数	4,003 件
命令網羅率	86.3%
欠陥数	9 件

表 2　対象とする欠陥
Table 2　Subject faults.

欠陥 ID	ファイル	行番号	エラー内容
1	dir_c/src01.c	524–535	条件誤り
2	dir_c/src01.c	443，445	処理漏れ
3	dir_c/src01.c	507–508	冗長処理
4	dir_r/src07.c	292	条件誤り
5	dir_r/src03.c	309，311，312	処理順番誤り
6	dir_r/src06.c	216	処理漏れ
7	dir_r/src06.c	183，184	論理誤り
8	dir_r/src07.c	216–234	冗長処理
9	dir_r/src10.c	266	マクロ誤り

プログラムに対し実行する．

移行後のプログラムとテストの概要を**表 1** に示す．テストは関数単位やファイル単位では存在せず，1 つの API を通じてライブラリ全体に対して行われる．

表 2 に欠陥局所化によって特定する欠陥をあげる．ここでは便宜上「欠陥」という表現としているが，文献 [9] に示したように，移行前後における非互換であり，許容される非互換も存在するため，すべてが修正されるものではない．また，これらは対象システムの開発者によって妥当性が確認されたものであるが，修正方法については開示されてないため，これらがすべての欠陥ではない可能性もある．

3.4　評価指標

欠陥局所化技術は全命令を疑惑値の降順に並べることで，順位を付けることができる．評価指標を考える場合，順位だけでは異なる大きさのプログラム間の比較が困難となるため，統一的な評価指標として，EXAM スコア [11]，LIL [2]，T-Score [12]，Expense [1] などの方法が提案されている．広く用いられているのは EXAM スコアで，n を欠陥のある命令の順位，N を全命令数としたとき，$\frac{n}{N}$ で求められる．これは，ユーザが疑惑値の降順に欠陥が含まれる命令かを調べたときに，全体の命令の何パーセントを調べれば欠陥を見つけることができるか，という考え方に基づく．これらの評価方法は，疑惑値が同スコアの場合と，複数行の欠陥の場合に，それぞれ問題が生じる．

3.4.1　同スコアの命令が複数の場合の扱い

当該要素と他候補要素の疑惑値が同じ場合，順位は並ぶこととなる．その場合の順位として，順位の平均を使うことが多いが，本論文では $E_{inspect}$ [13] を用いる．$E_{inspect}$ は，順位の平均の問題を解決するために考案された．たとえば同スコアの要素がすべて欠陥だった場合に順位を平均とすると，本来ユーザはそれらのどの要素でも早期に確認できるにもかかわらず不当に低い順位になってしまう．

同スコアの要素の個数を n，そのうち欠陥である要素の個数を n_f，同スコアの要素の開始位置を P_{start} としたとき，$E_{inspect}$ は以下の式で定義される．

$$E_{inspect} = P_{start} + \sum_{k=1}^{n-n_f} k \frac{\binom{n-k-1}{n_f-1}}{\binom{n}{n_f}}$$

$E_{inspect}@n$ は $E_{inspect}$ で順位付けしたときに，上位 n 位以内に存在する欠陥数を意味する．

3.4.2　複数行の欠陥の扱い

複数行にわたる欠陥箇所は，それぞれの行で異なる順位となる場合がある．文献 [7] では以下の 3 つのシナリオで評価方法を考えている．

(1) 最良ケース：いずれかの欠陥行が特定される必要がある
(2) 最悪ケース：すべての欠陥行が特定される必要がある
(3) 平均ケース：50%の欠陥行が特定される必要がある

これらは欠陥行が 1 行の場合にはすべて同じ値となる．本論文では特に記載がない場合は最良ケースを使う．

4.　評価結果

3 章で示した方法に基づいて各 RQ に関する調査を行った結果を示す．

4.1　RQ1：各ミューテーション解析の実行時間はどのくらいか？

表 3 に SDL のみを用いた MBFL（SDL-MBFL）と，SDL を含む一般的なミューテーションオペレータを用いた MBFL（MBFL）と，SDL 以外のミューテーションオペレータを用いた MBFL（MBFL w/o SDL）の実行結果を示す．表中では文献 [14] における表記に合わせ，命令削除ミューテーションオペレータを SDL ではなく SSDL とし

© 2020 Information Processing Society of Japan

ている．

SDL-MBFL の実行時間は MBFL に比べ 20.3%減となっている．これは，SDL-MBFL がミューテーションオペレータとしては 1 種類だけの利用にもかかわらず，実行ミュータント数は MBFL のミュータント数の 65.5%にも及ぶためである．

4.2 RQ2：SDL-MBFL の疑惑値の算出式はどれが良いか？

SDL-MBFL の疑惑値の算出式を変えたときの結果を図 **6** と表 **4** に示す．

表 **3**　ミューテーション実行結果
Table 3　Results of mutation analysis.

	SDL-MBFL	MBFL	MBFL w/o SDL
実行時間 (hours)	150.8	189.3	38.5
ミュータント数	2,734	4,172	1,438
延べ実行テストケース数	5,016,143	8,089,454	3,073,311
ミューテーションスコア	22.17%	26.01%	33.31%
ミューテーションオペレータ	SSDL	OAAN, OBBN, ORRN, OSSN, OAAA, OBBA, OSSA, SSDL	OAAN, OBBN, ORRN, OSSN, OAAA, OBBA, OSSA

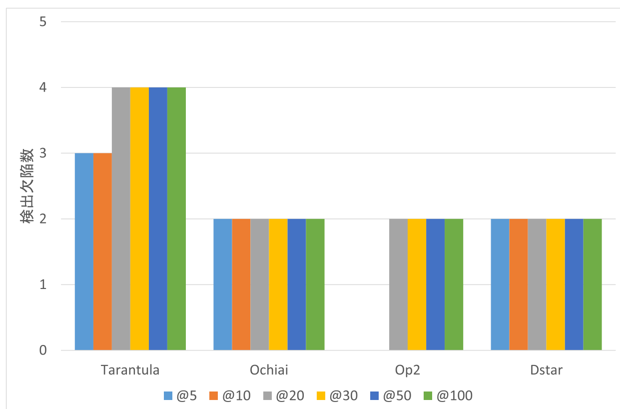


図 **6**　SDL-MBFL の各疑惑値算出式における $E_{inspect}$@n
Fig. 6　$E_{inspect}$@n in each of the SDL-MBFL suspiciousness calculation formulas.

これらのなかで Tarantula は $E_{inspect}$ においても EXAM スコアにおいても最も優れていた．また，Ochiai，DStar の間には $E_{inspect}$，EXAM スコアともにほとんど差はなく，Op2 が最も悪い結果となった．

4.3 RQ3：SDL-MBFL は他の欠陥局所化手法と比較して欠陥箇所を上位にあげられるか？

SDL-MBFL と他の欠陥局所化手法と比較した結果を図 **7** と表 4 に示す．

SDL-MBFL は SBFL や SDL なしの MBFL よりも 100 位以内においてより多くの欠陥が見つけられた．一方で MBFL は SDL-MBFL と同等もしくは若干劣る程度であり，ミューテーションオペレータに SDL を含むか否かが欠陥局所化性能へ支配的な影響を持つことが分かる．EXAM スコアは SBFL や SDL なしの MBFL の方が優れていた．

4.4 RQ4：SDL-MBFL と SBFL のハイブリッド手法は欠陥箇所を上位にあげられるか？

SDL-MBFL と SBFL のハイブリッド手法を比較した結果を図 **8** と表 4 に示す．

$E_{inspect}$@20 において，SDL-MBFL は他のハイブリッド

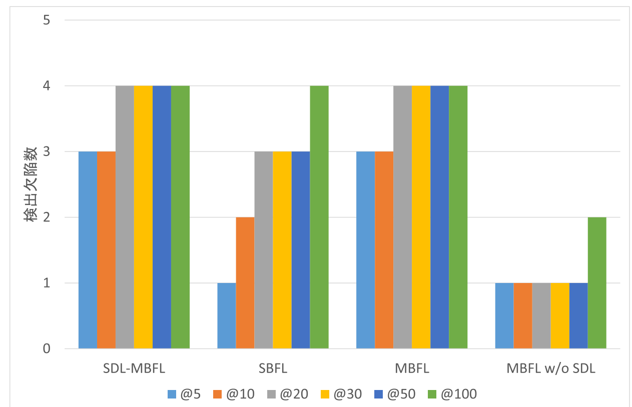


図 **7**　各欠陥局所化手法の $E_{inspect}$@n
Fig. 7　$E_{inspect}$@n for each fault localization technique.

表 **4**　欠陥局所化の結果（$E_{inspect}$ による順位，平均 EXAM 値）
Table 4　Results of fault localization (rank and average EXAM by $E_{inspect}$).

欠陥 ID	SDL-MBFL (Tarantula)	SDL-MBFL (Ochiai)	SDL-MBFL (Op2)	SDL-MBFL (DStar)	SBFL	MBFL	MBFL-w/o-SDL	Hybrid-Failover	Hybrid-Average	Hybrid-Max
13.0	3.3	10.2	3.3	9.0	19.0	634.2	23.2	7.0	25.2	
1.3	634.0	663.0	663.0	181.0	1.9	634.2	2.2	48.0	2.4	
1.3	3.3	10.2	3.3	102.0	1.9	284.0	2.2	29.0	2.4	
643.5	645.5	658.5	658.5	604.0	788.5	301.0	1535.5	904.0	866.0	
645.0	1645.0	2230.0	1645.0	108.0	262.0	81.0	2562.2	620.0	312.0	
515.5	254.0	183.5	237.8	70.0	617.7	634.2	1248.8	302.5	283.0	
515.5	254.0	183.5	237.8	13.5	363.0	133.0	1248.8	80.5	87.5	
1.3	447.0	428.0	428.0	2.2	1.9	4.0	2.2	1.8	2.4	
645.0	1645.0	2230.0	1645.0	1537.0	1991.5	634.2	1509.0	2281.5	2274.0	
EXAM	0.101	0.112	0.133	0.112	0.053	0.082	0.068	0.164	0.086	0.078

© 2020 Information Processing Society of Japan

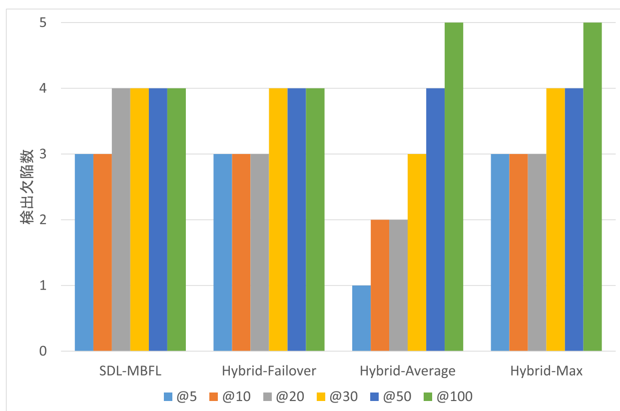


図 8　各ハイブリッド欠陥局所化手法の $E_{inspect}$@n
Fig. 8　$E_{inspect}$@n for each hybrid fault localization technique.

手法より優れるが，$E_{inspect}$@100 や EXAM スコアについては，Hybrid-Average や Hybrid-Max の方が優れているという結果が得られた．総合的な観点では，Hybrid-Max が他のハイブリッド手法より良い結果となっている．

5.　議論

5.1　実用における実行コスト対効果

欠陥局所化技術の実行時間は実用において大切な観点である．本論文ではミューテーションオペレータを SDL のみに限定することで，通常の MBFL と同等の欠陥局所化性能を維持しながら実行時間を約 20%削減できた．一方で，SBFL は MBFL とは異なり 1 回のテスト実行のみで疑惑値を計算できるため，数秒で実行が完了し，欠陥局所化性能は，100 位以内の欠陥数を示す $E_{inspect}$@100 においては SDL-MBFL に劣るが，平均を示す EXAM スコアでは他の手法を上回った．この節では実用において，これらの特性を考慮して欠陥局所化技術をどのように活用していけばよいかについて議論する．

一般的にミューテーション解析は時間がかかるため，MBFL の実行コストに対する効果については既存研究 [7], [13], [15] でも議論されてきている．また，文献 [16] では欠陥局所化の結果を得るまでに 1 時間以上待てる実務家は 9%以下であり，逆に 1 分以内であれば 90%以上が満足できるという調査結果を示している．

実用において実行時間の問題を緩和する方法の 1 つとして，継続的インテグレーション（CI）の中の自動テスト時に組み込み，テストの失敗をトリガとして欠陥局所化を実行する方法が考えられる．この方法により，欠陥局所化は開発者の作業を邪魔することなくバックグラウンドで処理され，ユーザは欠陥局所化の完了時に CI からの通知により欠陥局所化の結果を入手することができる．文献 [16] においても，何人かの実務家から CI に組み込むことについて好意的なコメントが寄せられている．同様の方法は，自動プログラム修正 [17] においても提案されている．

また，成功テストケースのみが通る命令に関するミュータントを省くことや，テストケース選択やミュータント削減などの実行時間削減の技術もさらなるスケーラビリティのために必要と考える．一方でテストやミュータントを削減することにより欠陥局所化性能を損なう恐れがある場合は，実用において許容できる範囲を見極めることが必要になる．本論文における実験では，関数呼び出しなどの単一の命令だけではなく，if 文などの複数行の命令に対しても削除のミューテーションを行ったため，より多くの時間がかかっている．これらのミュータントの削減が難しいのは，欠陥局所化性能への影響が大きいためである．5.3 項でも述べるが，これらの代替となり，かつ，よりミュータント数が減らせるミューテーションオペレータを探索することは今後の課題である．

5.2　欠陥の特徴

本論文では，9 つの欠陥を一度に局所化する評価を行った．複数の欠陥に対する SBFL は，1 カ所に対する SBFL と比較して困難であることが知られている [18], [19]．主な原因としては，それぞれの欠陥が相互に影響を及ぼし合い，テスト結果と欠陥との関係を複雑にするためである．たとえば，1 つの欠陥によってメモリ上のデータに誤りが生じたとしたときに，出力まで伝搬する途中で他の欠陥によってそのデータの誤りが書き換えられ隠されてしまうことがある．また，3.3 節で記述したように，本論文の評価対象の欠陥はシステム再構築前後における非互換であり，修正が不要な非互換も存在するため，また，対象システムの開発者による修正方法が開示されてないため，欠陥の修正を 1 つずつ行うシナリオを取ることができない．このように複数の欠陥を並行してデバッグするようなシナリオに対しては，クラスタリングによる手法 [20] や，整数計画法を用いる手法 [21] などが提案されている．

MBFL の 1 つである MUSE [2] では，複数の欠陥が存在する場合に対しても局所化に成功していることを実験的に示している．これはミューテーションによって該当箇所の命令が出力へ与える影響を，それぞれの命令ごとに独立して調査することができるためであると考えられる．

SBFL では上位にあげられないが，MBFL では上位となる欠陥は，欠陥 ID2 と 3 である．欠陥 ID1 も含め，これらは同じ関数内に存在し，欠陥 ID2 の後の処理に欠陥 ID3，欠陥 ID3 の処理の後に欠陥 ID1 が存在する．Listing 1 は欠陥 ID2，Listing 2 は欠陥 ID3 のそれぞれのソースコード箇所に理解の妨げにならない程度の変更（変数名など）を加えたものである．

欠陥 ID2 は，移行前のプログラムでは `isPart` の値にかかわらず 447 行目から 451 行目の処理を行っていたため，移行後のプログラムにおいて `isPart` が 0 の場合に処理を行わないことが非互換となってしまっていた．テス

© 2020 Information Processing Society of Japan

Listing 1 欠陥 ID2

```
442 switch (isPart) {
443 case 0:
444     /* no partial size check */
445     break;
446 case 1:
447     if (! ((Partial_size == 0) ||
448         (Partial_size >= PART_SIZE_MIN &&
449          Partial_size <= PART_SIZE_MAX))) {
450         return -1;
451     }
452     break;
```

Listing 2 欠陥 ID3

```
507 if (! (port >= 0 &&
508        port <= 65535)) {
509     return -1;
510 }
```

トとしては，isPart が 0 のときに Partial_size の値によって成否が分かれるが，成功テストケースが失敗テストケースよりも多いため，SBFL では局所化が困難であった．SDL-MBFL では命令削除ミューテーションにより break 文を削除した場合に fall through となり，isPart が 0 のときは 1 のときと同じ振舞いとなる．つまり，これは移行前と同じ処理となるため，SDL-MBFL は適切な局所化が可能となった．

欠陥 ID3 は，移行前のプログラムでは port の値のチェックを行っていなかったため，移行後のプログラムでチェックされるよう改善されたが，それが非互換となっていた．分岐の条件文の誤りは，成功テストケースと失敗テストケースの両方が通るため，SBFL では局所化されにくい．SDL-MBFL では 507 行目からの if 文全体を削除するミューテーションによって，移行前のプログラムと同等の振舞いとなるため，局所化が成功した．

一方，欠陥 ID5，6，7 は SBFL と比較して SDL-MBFL の方が大きく欠陥局所化性能が劣る．なかでも ID7 の $E_{inspect}$ は SBFL で 13.5 と高性能である一方，SDL-MBFL では 515.5 となっており，両者の得意・不得意の特徴を最も表している欠陥と見なせる．Listing 3 に，理解の妨げにならない程度の変更を加えた，欠陥 ID7 のソースコードを示す．移行前のプログラムでは，smtp_auth() の結果にかかわらず，smtp_data() が失敗した場合のエラーコードをそのまま返していたが，移行後のプログラムでは，smtp_auth() で失敗し，かつ，smtp_data() でも失敗した場合は，smtp_auth() でのエラーコードを返却するため，非互換が生じていた．SDL-MBFL では 184 行目の命令を削除した場合のテスト結果の影響を測定するが，180 行目ですでに error の値を auth_error に設定しているため，命令を削除しても出力が変わらないことが，局所化を困難にしている原因である．

Listing 3 欠陥 ID7

```
179 if ((rc = smtp_auth()) < 0) {
180     auth_error = error = rc;
181 }
182 if ((rc = smtp_data()) < 0) {
183     if(auth_error != 0){ /* authentication error */
184         error = auth_error;
185     }else{
186         error = rc;
187     }
188     return error;
189 }
```

本実験では，Metallaxis をベースとして SDL-MBFL を実装したが，その理由の 1 つとして，命令削除ミューテーションのみによって失敗テストケースを成功に変化させられることは直感的には多くなく，MUSE は適していないと考えたためであった．しかし，欠陥 ID2，3 は失敗テストケースが成功に変化したため，これらについては MUSE をベースにした SDL-MBFL によっても上位にあげられた可能性がある．

5.3 ミューテーションオペレータの選び方

欠陥 ID7 の局所化の様子を観察すると，ミューテーションオペレータの選び方の工夫が必要なことが分かる．[14] における SSDL のようにすべてのタイプの命令に対して削除するよりも，欠陥 ID2，3 のような 1 行単位の命令に対する削除と，欠陥 ID7 のような if 文の条件に対しての削除（つまりつねに真になる）が有効ではないかと考える．これは Pit [22] における Void Method Call Mutator，Remove Conditionals Mutator を組み合わせたものに近い考え方である．SSDL では if 文，for 文などの複合命令に対しても削除するが，そのような削除は影響範囲が大きく，局所化を困難にする可能性がある．また，そのような削除は，if 文内の条件に対しての影響を測ることができないため，局所化において不利に働くことがある．ただし，観察した例はまだ多くなく，一般性についてはさらなる調査が必要と考える．

6. 関連研究

6.1 ミューテーションオペレータの選択

ミューテーションオペレータの選択は，実行するミュータントの数を減らすことでミューテーション解析の実行時間を短縮することを目的とする．

Offutt ら [23] は 6 種類のミューテーションオペレータで元の 99.5%の精度でミューテーションスコアを計測できることを示した．Barbosa ら [24] はミューテーションオペレータ選択の指針を示し，65.02%に削減されたミュータントによって元の 99.6%の精度のミューテーションスコアを実現した．

© 2020 Information Processing Society of Japan

Deng ら [4] は命令削除ミューテーションオペレータによって Java におけるテスト評価に有効なことを示した．Delamaro ら [5] は命令削除ミューテーションオペレータ単一ですべてのオペレータを使った場合と同様の効果があることを示した．これらはミューテーション解析における命令削除ミューテーションオペレータの効果を測定したもので，欠陥局所化における効果は測定されていない．

6.2　ミューテーションベース欠陥局所化

2.2.2 項で MBFL の代表的な手法として MUSE と Metallaxis を紹介したが，それらを基にさらなる改良を加えた手法も存在する．

Li と Zhang [25] は，テストケース単位ではなくアサーション単位でミューテーションがプログラムに与える影響を記録し，それらの情報を学習することで欠陥がありそうな箇所の順位付けを行う手法 TraPT を提案した．Defects4J を用いた実験では既存の MBFL に比べ，TraPT はより高い欠陥局所化性能を示した．

Gong ら [26] はミュータントとテストケースの選択手法 Dynamic Mutation Execution Strategy（DMES）を提案した．初めに失敗テストケースのみで各ミュータントの上限疑惑値を計算し，その値が一定以上のミュータントのみを選択する．さらにあるミュータントの成功テストケースの実行中に，殺せる成功テストケース数が一定割合以上を超えたときに，そのミュータントにおいて残りの成功テストケースの実行をスキップする．

Lôbo de Oliveira ら [27] は，成功テストケースが欠陥局所化に大きく貢献しないことに着目し，失敗テストケースを通るミュータントのみ生成する手法 FTMES を提案した．Defects4J を用いた評価では，DMES などの既存手法よりも高速かつ高精度であることことを示した．

ここで紹介した手法はミューテーションオペレータを限定した場合でも採用することができるため，SDL-MBFL の改善に活用できる可能性がある．

6.3　産業用ソフトウェアに対するデバッグ手法の評価

Siemens suite は欠陥局所化の評価として最も多く用いられる産業用ソフトウェアであり，文献 [28] によると 90 件以上の論文で用いられている．しかし，小規模で，かつ，実際の欠陥ではなく，人工的に挿入された欠陥を対象としている．

産業用ソフトウェアに対し自動プログラム修正技術を適用した報告は何件か存在する．Naitou ら [29] は企業で開発された Java のプログラムと欠陥に対し，自動プログラム修正技術である jGenProg を適用することで，9 件の欠陥のうち，1 件を自動で修正できたことを報告した．池田ら [30] は企業内の実際の C 言語のプログラムと欠陥に対し，自動プログラム修正技術である Prophet を試行し，2 件中 1 件のパッチが得られたことを報告した．Noda ら [31] は企業内で 13 年以上開発・運用が続いている Java のソフトウェアにおいて，自動プログラム修正技術 Elixir で 20 件中 2 件のパッチ生成成功であったものを改善し，20 件中 8 件のパッチ生成に成功した．これらの報告において，自動プログラム修正の中で用いられる欠陥局所化は Ochiai による SBFL であり，欠陥局所化自体の評価については記述されていない．

7.　おわりに

本論文では命令削除ミューテーションオペレータを用いたミューテーションベース欠陥局所化を提案し，実際の製品となっている産業用ソフトウェアにおける 9 件の欠陥を用いて評価を行った．今回の対象の評価結果として，SDL-MBFL では Tarantula の疑惑値算出式で最も欠陥局所化性能が高く，また，高い順位での欠陥検出数が SBFL や SDL なしの MBFL よりも多かった．SBFL とのハイブリッドな手法では，疑惑値の最大値を選ぶ手法である Hybrid-Max が総合的に良い性能を示していた．実行時間に関しては MBFL と比較して 20.3%削減できたが，それでも長時間ではあるので実用するためには継続的インテグレーションに組み込むことなどの工夫が必要だという結論となった．

今後の課題としては，他の疑惑値算出方法や他のミューテーションオペレータを用いた MBFL との定量的な比較や，他のミュータント削減手法やテストケース削減手法との組合せによる実行時間削減がある．また，企業内のさらなるソフトウェアにおける評価についても取り組んでいく必要がある．

参考文献

[1] Jones, J.A. and Harrold, M.J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique, *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pp.273–282, Association for Computing Machinery (online), DOI: 10.1145/1101908.1101949 (2005).

[2] Moon, S., Kim, Y., Kim, M. and Yoo, S.: Ask the Mutants: Mutating Faulty Programs for Fault Localization, *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp.153–162 (online), DOI: 10.1109/ICST.2014.28 (2014).

[3] Papadakis, M. and Le Traon, Y.: Metallaxis-FL: mutation-based fault localization, *Software Testing, Verification and Reliability*, Vol.25, No.5-7, pp.605–628 (2015).

[4] Deng, L., Offutt, J. and Li, N.: Empirical Evaluation of the Statement Deletion Mutation Operator, *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp.84–93 (online), DOI: 10.1109/ICST.2013.20 (2013).

[5] Delamaro, M.E., Deng, L., Durelli, V.H.S., Li, N. and Offutt, J.: Experimental Evaluation of SDL and One-Op Mutation for C, *2014 IEEE 7th International Confer-*

ence on Software Testing, Verification and Validation, pp.203–212 (online), DOI: 10.1109/ICST.2014.33 (2014).

[6] Offutt, A.J. and Untch, R.H.: *Mutation Testing for the New Century*, Kluwer Academic Publishers, chapter Mutation 2000: Uniting the Orthogonal, pp.34–44 (2001) (online), available from ⟨http://dl.acm.org/citation.cfm?id=571305.571314⟩.

[7] Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D. and Keller, B.: Evaluating and Improving Fault Localization, *2017 IEEE/ACM 39th International Conference on Software Engineering* (*ICSE*), pp.609–620 (online), DOI: 10.1109/ICSE.2017.62 (2017).

[8] Tokumoto, S., Yoshida, H., Sakamoto, K. and Honiden, S.: MuVM: Higher Order Mutation Analysis Virtual Machine for C, *2016 IEEE International Conference on Software Testing, Verification and Validation* (*ICST*), pp.320–329 (online), DOI: 10.1109/ICST.2016.18 (2016).

[9] Tokumoto, S., Sakamoto, K., Shimojo, K., Uehara, T. and Washizaki, H.: Semi-automatic Incompatibility Localization for Re-engineered Industrial Software, *2014 IEEE 7th International Conference on Software Testing, Verification and Validation*, pp.91–94 (online), DOI: 10.1109/ICST.2014.20 (2014).

[10] Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs, *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, *OSDI'08*, pp.209–224, USENIX Association (2008) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1855741.1855756⟩.

[11] Wong, E., Wei, T., Qi, Y. and Zhao, L.: A Crosstab-based Statistical Method for Effective Fault Localization, *2008 1st International Conference on Software Testing, Verification, and Validation*, pp.42–51 (2008) (online), DOI: 10.1109/ICST.2008.65.

[12] Liu, C., Fei, L., Yan, X., Han, J. and Midkiff, S.P.: Statistical Debugging: A Hypothesis Testing-Based Approach, *IEEE Trans. Software Engineering*, Vol.32, No.10, pp.831–848 (online), DOI: 10.1109/TSE.2006.105 (2006).

[13] Zou, D., Liang, J., Xiong, Y., Ernst, M.D. and Zhang, L.: An Empirical Study of Fault Localization Families and Their Combinations, *IEEE Trans. Software Engineering*, p.1 (online), DOI: 10.1109/TSE.2019.2892102 (2019).

[14] Agrawal, H., DeMillo, R., Hathaway, R., 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, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University (1989).

[15] Liu, Y., Li, Z., Zhao, R. and Gong, P.: An Optimal Mutation Execution Strategy for Cost Reduction of Mutation-Based Fault Localization, *Inf. Sci.*, Vol.422, No.C, pp.572–596 (online), DOI: 10.1016/j.ins.2017.09.006 (2018).

[16] Kochhar, P.S., Xia, X., Lo, D. and Li, S.: Practitioners' Expectations on Automated Fault Localization, *Proc. 25th International Symposium on Software Testing and Analysis*, *ISSTA 2016*, pp.165–176, Association for Computing Machinery (online), DOI: 10.1145/2931037.2931051 (2016).

[17] Urli, S., Yu, Z., Seinturier, L. and Monperrus, M.: How to Design a Program Repair Bot? Insights from the Repairnator Project, *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track* (*ICSE-SEIP*), pp.95–104 (2018).

[18] Debroy, V. and Wong, W.E.: Insights on Fault Interference for Programs with Multiple Bugs, *2009 20th International Symposium on Software Reliability Engineering*, pp.165–174 (online), DOI: 10.1109/ISSRE.2009.14 (2009).

[19] DiGiuseppe, N. and Jones, J.A.: On the Influence of Multiple Faults on Coverage-Based Fault Localization, *Proc. 2011 International Symposium on Software Testing and Analysis*, pp.210–220, Association for Computing Machinery (online), DOI: 10.1145/2001420.2001446 (2011).

[20] Jones, J.A., Bowring, J.F. and Harrold, M.J.: Debugging in Parallel, *Proc. 2007 International Symposium on Software Testing and Analysis*, pp.16–26, Association for Computing Machinery (online), DOI: 10.1145/1273463.1273468 (2007).

[21] Högerle, W., Steimann, F. and Frenkel, M.: More Debugging in Parallel, *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp.133–143 (online), DOI: 10.1109/ISSRE.2014.29 (2014).

[22] Coles, H.: PIT, available from ⟨http://pitest.org⟩.

[23] Offutt, A.J., Rothermel, G. and Zapf, C.: An Experimental Evaluation of Selective Mutation, *Proc. 15th International Conference on Software Engineering*, *ICSE '93*, pp.100–107, IEEE Computer Society Press (1993) (online), available from ⟨http://dl.acm.org/citation.cfm?id=257572.257597⟩.

[24] 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*, Vol.11, No.2, pp.113–136 (online), DOI: 10.1002/stvr.226 (2001).

[25] Li, X. and Zhang, L.: Transforming Programs and Tests in Tandem for Fault Localization, *Proc. ACM Program. Lang.*, Vol.1, No.OOPSLA (online), DOI: 10.1145/3133916 (2017).

[26] Gong, P., Zhao, R. and Li, Z.: Faster mutation-based fault localization with a novel mutation execution strategy, *2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops* (*ICSTW*), pp.1–10 (online), DOI: 10.1109/ICSTW.2015.7107448 (2015).

[27] Lôbo de Oliveira, A.A., Gonçalves Camilo-Junior, C., Noronha de Andrade Freitas, E. and Rizzo Vincenzi, A.M.: FTMES: A Failed-Test-Oriented Mutant Execution Strategy for Mutation-Based Fault Localization, *2018 IEEE 29th International Symposium on Software Reliability Engineering* (*ISSRE*), pp.155–165 (online), DOI: 10.1109/ISSRE.2018.00026 (2018).

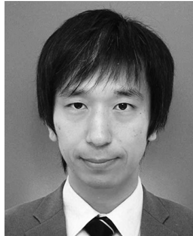
[28] Wong, W.E., Gao, R., Li, Y., Abreu, R. and Wotawa, F.: A Survey on Software Fault Localization, *IEEE Trans. Software Engineering*, Vol.42, No.8, pp.707–740 (online), DOI: 10.1109/TSE.2016.2521368 (2016).

[29] Naitou, K., Tanikado, A., Matsumoto, S., Higo, Y., Kusumoto, S., Kirinuki, H., Kurabayashi, T. and Tanno, H.: Toward Introducing Automated Program Repair Techniques to Industrial Software Development, *Proc. 26th Conference on Program Comprehension*, pp.332–335, Association for Computing Machinery (online), DOI: 10.1145/3196321.3196358 (2018).

[30] 池田 翔，中野大扉，亀井靖高，佐藤亮介，鵜林尚靖，吉武浩，矢川博文：企業内ソースコードに対する自動バグ修正

© 2020 Information Processing Society of Japan

技術適用の試み，信学技報，Vol.118, No.471, pp.193–198 (2019).
[31] Noda, K., Nemoto, Y., Hotta, K., Tanida, H. and Kikuchi, S.: Experience Report: How Effective Is Automated Program Repair for Industrial Software?, *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering* (*SANER*), IEEE (2020).



徳本 晋　（正会員）

2002年早稲田大学理工学部電子・情報通信学科卒業．2004年東京大学大学院情報理工学系研究科コンピュータ科学専攻修士課程修了．富士通（株）を経て2009年より（株）富士通研究所に勤務，現在に至る．2017年東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程単位取得満期退学．2018～2020年度早稲田大学招聘研究員．2019～2020年度国立情報学研究所外来研究員．ソフトウェアテストやデバッグに関する研究に従事．

本位田 真一　（正会員）

1978年早稲田大学大学院理工学研究科修士課程修了．（株）東芝を経て2000年国立情報学研究所教授，2012年同研究所副所長を兼務．2001年東京大学大学院情報理工学系研究科教授を兼任，2018年より早稲田大学理工学術院教授，現在に至る．現在，英国UCL客員教授ならびに国立情報学研究所GRACEセンター長を兼任．2005年度パリ第6大学招聘教授．2015年度リヨン第1大学招聘教授．日本学術会議連携会員．本会フェロー．

© 2020 Information Processing Society of Japan