ソースコード記述に着目した Fault Localizationに対する適合性の提案

佐々木 唯[1,2,a)]　肥後 芳樹[1,b)]　柗本 真佑[1,c)]　楠本 真二[1,d)]

概要：ソフトウェア開発におけるデバッグ作業支援として，Spectrum-Based Fault Localization（SBFL）に関する研究が盛んに行われている．SBFL は，テストケースごとの成否と，どの文が実行されたかという実行経路情報を用いて，プログラム中の欠陥箇所を推測する手法である．高水準言語にはさまざまな記述方法が用意されており，ある機能を実現するためのソースコードの記述方法は幾通りも存在する．そのため，その記述方法の違いが SBFL の精度に差を生じさせている可能性があると著者らは考えた．そこで本論文では，プログラム自体が SBFL にどの程度適しているかという特性（SBFL 適合性）を持つと考え，SBFL 適合性の計測手法を提案する．提案手法は，欠陥を含まないプログラムに対し，ミューテーションテスト技術を活用して疑似的な欠陥を発生させ，SBFL によってその疑似的な欠陥をどの程度正確に検出できたかを計測する．プログラム構造の違いによる SBFL 適合性の違いについて，リファクタリングを題材に確認したところ，SBFL 適合性を向上させるプログラム構造の特徴を発見した．

1. はじめに

ソフトウェア開発において，デバッグは多大な労力とコストを必要とする作業である．デバッグ作業がソフトウェア開発コストの過半数を占めるという報告もある [1,2]．このため，デバッグ支援に関する研究が盛んに行われている．

デバッグ支援の研究分野の 1 つに欠陥限局技術がある．欠陥限局とは，欠陥が含まれている箇所を推測する技術である．中でも近年，実行経路情報を用いた欠陥限局（Spectrum-Based Fault Localization，以降，SBFL）に関する研究が盛んに行われている [3]．SBFL は，失敗するテストケースによって実行される文は欠陥を含む可能性が高いという考えに基づき，テストケースごとの成否と，プログラム中で実行される文の情報（以下，実行経路情報）を用いて，欠陥を含む文を推測する技術である．SBFL 技術では，プログラム中の各文に対して欠陥を含む可能性（以降，疑惑値）を数値化する．欠陥を含む文の疑惑値が他の文と比べてより高い値であるほど，SBFL 結果がより正確であると言える．

ソフトウェアには様々な品質の観点が存在する．ISO/IEC 25010[*1]で定義されているソフトウェア製品の品質モデルでは，8 つの品質特性，及び各特性から派生する副特性が定義されている[*2]．品質特性の 1 つである Maintainability（保守性）の副特性に Analysability（解析性）がある．解析性とは，ソフトウェアの不具合の原因を診断することや，修正すべき箇所を識別することに対する有効性や効率の程度を示す[*3]．

高水準言語にはさまざまな記述方法が用意されており，開発者は自身の好みや組織の方針に従って，必要な機能の実装方法を決定する．実装方法が変わるとテストケース毎の実行経路情報が変わる可能性があり，更に文の疑惑値やその順位も変わる可能性がある．従って，プログラム自体が SBFL を用いた欠陥箇所の特定にどの程度適しているかという特性を持っていると考えることができる．

そこで本論文では，あるプログラムに対する SBFL を用いた欠陥限局の結果がどの程度正確かを，プログラムの**SBFL 適合性**と定義する．これは，あるプログラムに対す

1 大阪大学大学院情報科学研究科 大阪府吹田市
2 株式会社日本総合研究所 東京都品川区
a) s-yui@ist.osaka-u.ac.jp
b) higo@ist.osaka-u.ac.jp
c) shinsuke@ist.osaka-u.ac.jp
d) kusumoto@ist.osaka-u.ac.jp

*1 ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models
*2 ISO/IEC 25010 に準ずる JIS X 25010 では，8 つの品質特性を機能適合性・性能効率性・互換性・使用性・信頼性・セキュリティ・保守性・移植性と定義している．
*3 ISO/IEC 25010 における Analysability の定義は以下の通り．Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

る SBFL 適用技術との親和性の一種であり，品質特性である保守性，及びその副特性である解析性に含まれる 1 つの観点とみなすことができる．ソフトウェアの品質特性の 1 つの観点として SBFL 適合性を扱うことにより，下記の活動が可能になる．

- 現在のプログラムに対する SBFL 結果がどの程度信頼できるかを事前に把握する．信頼できると判断された場合は，SBFL 技術を利用することにより欠陥限局を行い，その結果を利用して開発者がデバッグ作業を行えば良い．
- SBFL 適合性が低いプログラムに対し，SBFL 適合性が向上するようなプログラム変換を行う．

本論文では，SBFL 適合性の計測方法も提案する．具体的には，欠陥を含まない正常なプログラムに対して，ミューテーションテスト [4] の技術を活用し，様々な箇所に疑似的な欠陥を発生させ，SBFL を実行する．SBFL によってどの程度正確に疑似的な欠陥箇所を特定できたかを計測することにより，元のプログラムの SBFL 適合性を示す値（以降，SBFL スコア）を計測する．

本論文ではリファクタリングを題材に，プログラム構造の違いによる SBFL スコアの計測結果を比較した．その結果，同じ条件分岐先で実行される文の数が少ないほど SBFL スコアが向上することを確認し，SBFL スコアを向上させるプログラム構造の変換例を発見した．

本論文による貢献は以下の点である．

- プログラムに対する SBFL 適合性を定義
- ミューテーションテストを用いた SBFL 適合性の計測手法を提案
- リファクタリングを題材として，プログラム構造の違いによって SBFL 適合性が異なる事例を確認
- SBFL 適合性を向上させるプログラム構造の特徴と変換例を発見

2. 関連研究

プログラム中の欠陥箇所を推測する手法の 1 つとして，実行経路情報を用いた欠陥限局（以降，SBFL）に関する研究が盛んに行われている [3]．SBFL は，各テストケースの成否と実行経路情報を用いて，文ごとの疑惑値，すなわち欠陥である可能性を示す値を算出する．実行経路情報とは，各テストケースがプログラム中のどの文を実行したかという情報である．直感的には，失敗するテストケースが多く通過する文は，成功するテストケースが多く通過する文より欠陥を含む可能性が高いと考えることができる．これまでに多くの SBFL 手法が提案されている [5–7] が，Abreu らは複数の SBFL 手法を比較した結果，Ochiai [8] の手法が最も優れていると結論づけている [9]．

SBFL はテストケース毎の実行経路情報に基づくため，テストケースの内容に結果が左右される．失敗するテストケースを元に他のテストケースを生成することで，より効率的に SBFL を行う研究が行われている [10,11]．

また，テストスイートの欠陥検出能力を評価する方法として，ミューテーションテストに関する研究が行われている [4]．ミューテーションテストでは，欠陥を含まない正常なプログラムに対して意図的に変更を加えた大量のプログラムが生成される．変更が加えられたプログラムをミュータントと呼ぶ．各ミュータントに対してテストを実行した際，テストが失敗すれば，そのテストはミュータントの変更箇所，すなわち疑似的な欠陥を検出する能力があると判断できる．ミューテーションテストには様々なツールが提案されており [12–14]，企業のソフトウェアに対するミューテーションテストの実用性も報告されている [15]．

ミューテーションテストを SBFL に応用する研究も行われている．ミューテーションテストでは意図的に変更が加えられるため，その変更箇所，すなわち欠陥の混入箇所は明らかである．欠陥の箇所が未知であるプログラムと，そのプログラムから生成された特定のミュータントが，共通のテストケースで失敗する場合，欠陥の箇所も共通である可能性が高い．この考えに基づき，欠陥限局の精度を向上させる手法が提案されている [16,17]．

3. SBFL 適合性

著者らは，プログラム自体が SBFL にどの程度適しているかという特性を持っていると考える．本論文では，この特性を **SBFL 適合性**と定義する．プログラムの機能やテストスイートが共通であっても，構造が異なることで SBFL を用いた欠陥限局の精度に違いが生じることがある．本章では，プログラム構造の違いによる SBFL 適合性の違いについて，具体的事例を用いて説明する．

図 1 は，2 つの入力値を受け取り，少なくとも一方が正の数であれば true を，そうでなければ false を返す処理を，2 通りの記述で表している．図 1(a) では結果を変数 result に代入して，最後にまとめて結果を返しているが，図 1(b) では結果が確定したタイミングで逐次返している．図 1(b) は図 1(a) に対して，リファクタリングが適用された状態である．このように return 文によって早く処理を抜ける書き方は early return と呼ばれる [18]．

このプログラムのテストスイートには図 1(c) を用意した．図 1 の各プログラムは同じ条件式に欠陥を含んでいる．図 1(a) の s_4 と図 1(b) の s'_4 は変数 b が 0 である場合も true を返してしまうため，テストケース t_4 は失敗する．

図 1(a), 1(b) の右側には，各テストケースの結果（P: 通過（成功），F: 失敗）と実行経路情報，及び Ochiai の計算式により算出した文の疑惑値（suspiciousness，図では susp

	プログラム (入力: a, b)	*susp*	t_1	t_2	t_3	t_4
	s_1: **boolean** result = **false**;	0.50	✓	✓	✓	✓
	s_2: **if** (0 < a)	0.50	✓	✓	✓	✓
	s_3: result = **true**;	0.00	✓	✓		
🐞	s_4: **if** (0 <= b) // correct: 0 < b	0.50	✓	✓	✓	✓
	s_5: result = **true**;	0.50	✓	✓	✓	✓
	s_6: **return** result;	0.50	✓	✓	✓	✓
	テスト結果：		P	P	P	F

(a) リファクタリング前

	プログラム (入力: a, b)	*susp*	t_1	t_2	t_3	t_4
	s'_2: **if** (0 < a)	0.50	✓	✓	✓	✓
	s'_3: **return true**;	0.00	✓	✓		
🐞	s'_4: **if** (0 <= b) // correct: 0 < b	0.71			✓	✓
	s'_5: **return true**;	0.71			✓	✓
	s'_6: **return false**;	-				
	テスト結果：		P	P	P	F

(b) リファクタリング後

テストケース	入力 (a, b)	期待値	実際の値
t_1:	(1, 1)	true	true
t_2:	(1, 0)	true	true
t_3:	(0, 1)	true	true
t_4:	(0, 0)	false	true

(c) テストスイート

図 1: 構造の異なる 2 つのプログラムの SBFL 結果

と記載）も記載している[*4]．疑惑値の取りうる値は 0 以上 1 以下であり，1 が最も疑惑が強いことを示す．図 1 より，欠陥を含む文 s_4, s'_4 は異なる疑惑値であると確認できる．

欠陥箇所が未知である状態において，SBFL 結果を元に欠陥の箇所を特定しようとする作業を考える．文ごとの疑惑値が算出されているため，作業者は疑惑値の高い順に文を確認する．図 1(a) の場合，欠陥を含む文の疑惑値は最も高いものの，同じ疑惑値を持つ文が計 5 つ存在するため，最大で 5 つの文を確認しなければ発見できない．一方で，図 1(b) であれば，2 つの文を確認すれば欠陥箇所を発見することができる．従って，図 1(a) に比べて図 1(b) の方が SBFL 結果の精度が高いと言える．

4. SBFL 適合性の計測方法

本章では，プログラムの SBFL 適合性の計測方法について述べる．SBFL 適合性を計測するための基本的なアイデアは，与えられたプログラムに対して意図的に変更を加えたプログラムを複数生成し，その変更箇所を欠陥とみなして，SBFL でどの程度正確に特定できるか計測することである．プログラムに意図的に変更を加える方法として，ミューテーションテストを活用する．本来ミューテーションテストとは，テストスイートがミュータントをどの程度正確に欠陥として検出できるかを計測するが，本手法は，ミュータントに含まれる疑似的な欠陥箇所を，SBFL がどの程度正確に識別できるかを計測する．

[*4] 用意したテストスイートでは図 1(b) の文 s'_6 は実行されないため，文 s'_6 の疑惑値の情報は存在しない．

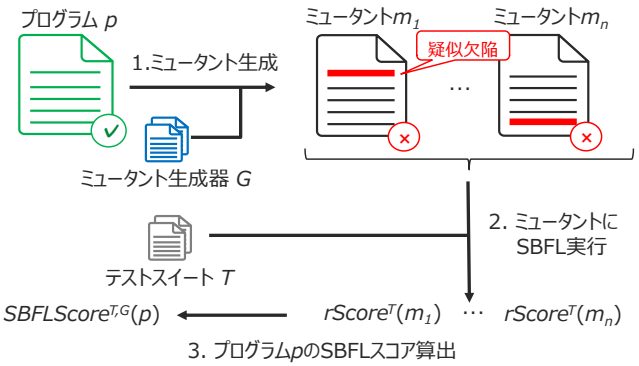

図 2: SBFL スコア算出の流れ

4.1 SBFL 適合性を構成する要素

SBFL 適合性はプログラム p 以外に次の 2 つの要素から構成される．

- テストスイート T
- ミュータント生成器 G

テストスイートに含まれるテストケースの内容によって SBFL の精度に違いが生じる可能性がある．例として，図 1(b) においてテストケースが t_3, t_4 のみである場合を考える．この 2 つのテストケースの実行経路情報は同一であるため，実行経路情報から計算される疑惑値は，少なくとも実行される文 s'_2, s'_4, s'_5 において同じ値となる．元のテストスイートによる SBFL 結果であれば，文 s'_2 より文 s'_4, s'_5 の方に欠陥が含まれている可能性が高いと判断できる．

ミューテーションテストにおいて，ミュータントを生成するためのプログラムの変換ルールをミューテーション演算子と表現する．本論文では，ミュータント生成のために用いるミューテーション演算子の集合をミュータント生成器と呼ぶ．SBFL 適合性はミュータント生成器に依存する．

4.2 算出方法

SBFL 適合性を示す値を **SBFL スコア**と定義する．SBFL スコアは 0 以上 1 以下の値をとり，値が高いほど SBFL 適合性が高いものとする．プログラム p の SBFL スコアは，テストスイート T，ミュータント生成器 G に依存するとして，$SBFLScore^{T,G}(p)$ と定義する．

SBFL スコアの算出の流れを図 2 に示す．SBFL スコアは次の 3 つのステップによって算出される．

(1) プログラム p から複数のミュータントを生成
(2) 各ミュータントに SBFL を実行し，変更箇所（疑似欠陥）の疑惑値の順位を算出
(3) 各ミュータントに含まれる疑似欠陥の疑惑値の順位から，SBFL スコアを算出

ステップ 1. ミュータント生成

プログラム p にミュータント生成器 G を用いて，ミュー

タントを生成する．このとき，各ミュータントは元のプログラムに対して一文だけが変更されるよう生成する．この変更が加えられた箇所を欠陥として扱うが，この時点ではテストを実行していないため，変更箇所が本当に欠陥となるかどうかは分からない．

ステップ 2. ミュータントに SBFL を実行

生成された各ミュータントに対して，テストスイート T を用いて SBFL を実行する．T に含まれる全てのテストケースを通過したミュータントは，欠陥を含まないミュータントであるとして対象から除外し，残ったミュータントの集合を $M^G(p)$ とする．$M^G(p)$ に含まれる各ミュータントに対し，文ごとの疑惑値を算出し，順位付けを行う．ここで，あるミュータント $m \in M^G(p)$ に含まれる各文 s について，以下を定義する．

- $susp^T(s)$: 文 s の疑惑値
- $rank^T(s)$: 文 s の疑惑値の順位
- $rScore^T(s)$: 文 s の疑惑値の正規化順位

疑惑値の算出には Ochiai の計算式を用いる．テストスイート T を実行したときのある文 s の疑惑値 $susp^T(s)$ は，Ochiai の計算式によって以下の通り定義される．以下の式において，$fail^T(s)$ は文 s を実行した失敗テストケースの数，$pass^T(s)$ は文 s を実行した成功テストケースの数，$totalFail^T$ は失敗テストケースの総数である．

$$susp^T(s) = \frac{fail^T(s)}{\sqrt{totalFail^T * (fail^T(s) + pass^T(s))}} \quad (1)$$

次に疑惑値の順位 $rank^T(s)$ を算出する．ある文の疑惑値の順位は，疑惑値の高い順に文を並べた際最大で確認しなければならない文の総数とする．例えば，疑惑値 1.0 の文が 2 つ，0.9 の文が 1 つ存在する場合は，疑惑値 1.0 の文はどちらも 2 位と扱い，疑惑値 0.9 の文は 3 位とする．

疑惑値の順位は，順位の母集団となる文の総数により異なる価値を持つ．例えば，10 個の文のうちの 10 位と，100 個の文のうちの 10 位とでは，後者の方が順位としての価値が高い．そこで，各文が文全体の中でどの程度上位に登場するかを示すため，順位を 0 以上 1 以下の範囲に線形に正規化する．文 s の正規化順位 $rScore^T(s)$ は以下の通り定義する．1 が最も順位が高く，0 が最も順位が低いことを示している．以下の式において，$totalStatements^T$ はテストスイート T によって実行される文の数である．

$$rScore^T(s) = 1 - \frac{rank^T(s) - 1}{totalStatements^T - 1} \quad (2)$$

また，ミュータント m に含まれる欠陥の疑惑値の正規化順位を $rScore^T(m)$ と定義する．各ミュータントに含まれる欠陥はただ 1 つの文であるため，この値はミュータント毎に一意である．ミュータント m の欠陥を含む文を $s^m{}_{fault}$ とすると，$rScore^T(m)$ は文 $s^m{}_{fault}$ の疑惑値の正規化順位である．

$$rScore^T(m) = rScore^T(s^m{}_{fault})$$

	プログラム (入力: a, b)	*susp*	*rank*	*rScore*
s_1:	**boolean** result = **false**;	0.50	5	0.20
s_2:	**if** (0 < a)	0.50	5	0.20
s_3:	result = **true**;	0.00	6	0.00
🐞 s_4:	**if** (0 <= b) // correct: 0 < b	0.50	5	0.20
s_5:	result = **true**;	0.50	5	0.20
s_6:	**return** result;	0.50	5	0.20

(a) リファクタリング前

	プログラム (入力: a, b)	*susp*	*rank*	*rScore*
s'_2:	**if** (0 < a)	0.50	3	0.33
s'_3:	**return true**;	0.00	4	0.00
🐞 s'_4:	**if** (0 <= b) // correct: 0 < b	0.71	2	0.67
s'_5:	**return true**;	0.71	2	0.67
s'_6:	**return false**;	-	-	-

(b) リファクタリング後

図 3: ミュータントへの SBFL 実行結果

ミュータントの $rScore$ が高いほど，そのミュータントの SBFL 結果の精度が高い，すなわち欠陥箇所を正確に特定できることを意味する．例として，図 1 のプログラムをミュータントと見立てた場合の SBFL 結果を図 3 に示す．欠陥を含む文 s_4, s'_4 の $rScore$，及びこのミュータント自体の $rScore$ はそれぞれ 0.20 と 0.67 である．3 章で述べた通り，リファクタリング後の方が SBFL 結果の精度が高いという事実を，$rScore$ によって示すことができている．

ステップ 3. SBFL スコアの算出

プログラム p から生成された各ミュータントの $rScore$ の平均値を SBFL スコアとし，以下の式で定義する．

$$SBFLScore^{T,G}(p) = \frac{1}{|M^G(p)|} \sum_{m \in M^G(p)} rScore^T(m)$$

5. 実験

Java を対象として提案手法を実装し，プログラム構造の違いにより SBFL 適合性がどのように変化するか確認するための実験を行った．

5.1 準備

本実験で用いるミューテーション演算子は，オープンソースのミューテーションテストツールである PIT [19] を参考とした．PIT の Web サイトではミューテーション演算子のカテゴリと内容が公開されている．PIT 自体はバイトコードとしてのミュータントしか生成しないため，ソースコードとしてミュータントを生成するよう，PIT のデフォルトカテゴリに含まれる 11 種類のミューテーション演算子を本実験のために実装した．本実験で用いるミューテーション演算子を表 1 に示す．本論文では表 1 の括弧内

表 1: 実験で用いるミューテーション演算子

ミューテーション演算子		説明	変換例	
			変換前	変換後
(CB)	Conditional Boundary	関係演算子の境界を変更する	`a<b`	`a<=b`
(INC)	Increments	インクリメントとデクリメントを入れ替える	`n++`	`n--`
(INV)	Invert Negatives	負の数を正の数に置き換える	`-n`	`n`
(MA)	Math	算術演算子を置き換える	`a+b`	`a-b`
(NC)	Negate Conditionals	関係演算子を置き換える	`a<b`	`a>=b`
(VM)	Void Method Calls	何の値も返さないメソッド呼び出しを削除する	`method();`	`;`
(PR)	Primitive Returns	プリミティブ型の戻り値を 0 に置き換える	`return 5;`	`return 0;`
(ER)	Empty Returns	戻り値の型に応じて空を表す値に置き換える	`return "str";`	`return "";`
(FR)	False Returns	戻り値を false に置き換える	`return true;`	`return false;`
(TR)	True Returns	戻り値を true に置き換える	`return false;`	`return true;`
(NR)	Null Returns	戻り値を null に置き換える	`return object;`	`return null;`

の表記を各ミューテーション演算子の略称として使用する．

本論文では，プログラム構造の違いとしてリファクタリングを題材とした．リファクタリングには様々なパターンが存在する [20] が，今回は，「条件式の簡素化」に分類される 8 種類のリファクタリングの中から表 2 に示す 5 種類のリファクタリングパターンを選定した．このカテゴリから選定した理由は，条件式の分岐によってテストケース毎の実行経路情報が変化するため，リファクタリングによって SBFL 結果が変化しやすいと考えたためである．また，本論文では SBFL を Java メソッド単位で実行し，対象となるメソッド内で文の順位を計算している．そのため，複数のメソッドにまたがるリファクタリングは対象外とした．

実験対象プログラムは，Fowler の書籍 [20] における各リファクタリングパターンのサンプルプログラムを参考に手作業で作成した．この際，ミューテーション演算子がなるべく多く適用されるよう配慮して作成した．例えば，条件式に boolean 型の変数だけを書くとミューテーション演算子が適用されないため，`var == true` のように関係演算子を用いて NC 演算子が適用されるようにしている．テストスイートは，プログラムから生成された全てのミュータントに対して，C2 カバレッジ（条件網羅）が満たされるよう手作業で作成した．

対象プログラムを図 4〜8 に示す．このうち，図 4〜6 は以降の節にて詳細を述べるため，生成されたミュータントに関する情報も掲載している．図 7, 8 は参考として本論文の末尾に掲載している．

5.2 結果と考察

5 種類の実験対象プログラムについて，リファクタリング前後それぞれのプログラムに対して提案手法を適用し，SBFL スコアを算出した．なお，いずれのプログラムにおいても，生成されたミュータントは少なくとも 1 つ以上のテストケースに失敗する，すなわち欠陥を含むミュータントであった．SBFL スコアの計測結果を表 2 の右側に示す．ケース 1〜3 はリファクタリング前の SBFL スコアが高く，ケース 4, 5 ではリファクタリング後の SBFL スコアが高いという結果となった．

例としてケース 5, 1, 3 のプログラムと生成されたミュータントに関する情報を図 4, 5, 6 に示す．(a) がリファクタリング前，(b) がリファクタリング後であり，それぞれの左側にプログラム，右側に各ミュータントにおける各文の *rScore* を表示している．*rScore* は数値と棒グラフで表示されており，各ミュータントのうち元のプログラムから変更された文，すなわち欠陥を含む文の *rScore* は太字で表現されている．この値は 4.2 節で述べた通りミュータントの *rScore* でもある．例えば，図 4(a) のミュータント m_1 は，文 s_2 に対して NC 演算子が適用されたミュータントであり，ミュータントの *rScore* が 0.67 であることを示している．太字で表現しているミュータントの *rScore* の平均値が *SBFLScore* である．

図 4 の文 s_2 と s'_2 のようにリファクタリング前後で互いに対応する文は同じ数字を添字として付与している．例えば図 4 の文 s_2 と s'_2 はリファクタリング前後で変化は無く，図 5 の文 s'_{1a} と s'_{1b} はリファクタリングによって文 s_1 から分割されたことを示している．また，図 6 の文 $s'_{\{5,7\}}$ は，文 s_5，s_7 がリファクタリングによって 1 つの文に集約されたことを示している．

また，図 4 の m_1 と m'_1 のように，互いに対応する文，すなわち同じ数字を添字に持つ文に同じミューテーション演

表 2: 対象リファクタリングと SBFL スコアの計測結果

ケース	リファクタリングパターン	SBFL スコア	
		前	後
1	条件記述の分解	0.81	0.53
2	条件式の統合	0.95	0.72
3	重複した条件記述断片の統合	0.69	0.53
4	制御フラグの削除	0.61	0.67
5	ガード節による入れ子条件記述の書き換え	0.83	0.95

		rScore							
	ミュータント:	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
ミューテーション演算子:		NC	CB	INV	NC	CB	INV	INV	PR
s_1: **int** result = 0;		0.67	0.50	0.50	0.33	0.33	0.33	0.33	0.67
s_2: **if** (x > 0)		**0.67**	**0.50**	0.50	0.33	0.33	0.33	0.33	0.67
s_3: result = -10;		0.50	1.00	**1.00**	0.00	0.00	0.00	0.00	0.17
s_4: **else if** (y > 0)		0.33	0.00	0.00	**1.00**	**0.83**	0.83	0.83	0.50
s_5: result = -20;		0.00	0.00	0.00	0.83	1.00	**1.00**	0.00	0.00
else		-	-	-	-	-	-	-	-
s_6: result = -30;		0.17	0.00	0.00	0.17	0.00	0.00	**1.00**	0.33
s_7: **return** result;		0.67	0.50	0.50	0.33	0.33	0.33	0.33	**0.67**

(a) リファクタリング前 (*SBFLScore*=0.83)

		rScore									
	ミュータント:	m'_1	m'_2	m'_3	m'_4	m'_5	m'_6	m'_7	m'_8	m'_9	m'_{10}
ミューテーション演算子:		NC	CB	INV	NC	CB	INV	INV	PR	PR	PR
s'_2: **if** (x > 0)		**1.00**	**0.75**	0.75	0.50	0.50	0.50	0.50	0.50	0.75	0.50
s'_3: **return** -10;		0.75	1.00	**1.00**	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00
s'_4: **if** (y > 0)		0.50	0.00	0.00	**1.00**	**0.75**	0.75	0.75	0.75	0.00	0.75
s'_5: **return** -20;		0.00	0.00	0.00	0.75	1.00	**1.00**	0.00	0.00	0.00	**1.00**
s'_6: **return** -30;		0.25	0.00	0.00	0.25	0.00	0.00	**1.00**	**1.00**	0.00	0.00

(b) リファクタリング後 (*SBFLScore*=0.95)

図 4: ケース 5: ガード節による入れ子条件記述の書き換え

算子が適用されて生成されたミュータントを，ミュータントのペアと呼び，$\langle m_1, m'_1 \rangle$ と表現する．ペアとなるミュータントの *rScore* を比較することで，同じ記述箇所に発生した欠陥の正規化順位，すなわちその欠陥箇所の特定のしやすさが，リファクタリング前後でどれだけ変化するかを確認することができる．

5.2.1 SBFL スコアが向上した例

例としてケース 5 について考察する．ケース 5 で適用した「ガード節による入れ子条件記述の書き換え」とは，後続処理の対象外となる条件が満たされれば return する処理を先頭に記述することで，早く処理を終了させるよう書き換えるリファクタリングである．3 章で用いたリファクタリング例と同様，このような書き方は early return とも呼ばれる．これらはソースコードのネストが深くなることを抑制する効果がある．

図 4 よりミュータントのペアを比較すると，$\langle m_1, m'_1 \rangle$, $\langle m_2, m'_2 \rangle$, $\langle m_8, m'_8 \rangle$ はリファクタリング後に *rScore* が向上，$\langle m_5, m'_5 \rangle$ はリファクタリング後に *rScore* が低下，それ以外は変わらなかった．リファクタリング前の各ミュータントにおいて，文 s_1, s_2, s_7 は同じ *rScore* であることに着目する．この文が欠陥箇所であるミュータントは m_1, m_2, m_8 であり，リファクタリング後に *rScore* が向上したミュータントのペアに含まれる．*rScore* の計算過程を確認したところ，これらの文の疑惑値もまた互いに同じ値であった．4.2 節で述べた通り，提案手法では同じ疑惑値を持つ文が多いほど，その文の順位が下がるような順位付けを行っている．文 s_1, s_2, s_7 はどのテストケースにおいても必ず実行される文であるため同じ疑惑値であったが，リファクタリング後は return 文が挿入されたことで，必ず実行される文が s'_2 のみとなったため，順位及び *rScore* が向上したと考えられる．

表 3: 条件分岐先ごとの文の分類（ケース 5）

(a) リファクタリング前

条件分岐先 C(s_2, s_4)	該当文
C(*, *)	$\{s_1, s_2, s_7\}$
C(T, *)	$\{s_3\}$
C(F, *)	$\{s_4\}$
C(F, T)	$\{s_5\}$
C(F, F)	$\{s_6\}$

(b) リファクタリング後

条件分岐先 C(s'_2, s'_4)	該当文
C(*, *)	$\{s'_2\}$
C(T, *)	$\{s'_3\}$
C(F, *)	$\{s'_4\}$
C(F, T)	$\{s'_5\}$
C(F, F)	$\{s'_6\}$

式 (1) において，実行されるテストケースが共通である文，すなわち同じ条件分岐先[*5]で実行される文は，常に同じ疑惑値を持つ．同じ条件分岐先で実行される文を明確にするため，条件分岐先ごとに文を分類した．その結果を表 3 に示す．この表において，C(s_2, s_4) は文 s_2 と s_4 にそれぞれ含まれる条件式の結果に応じた条件分岐先を意味する．条件式の結果のとりうる値は，$\{T, F, *\}$ の 3 通りであり，それぞれ真，偽，影響なしを示す．例えば C(*, *) は，いずれの条件式にも影響を受けない条件分岐先であり，リファクタリング前は文 s_1, s_2, s_7 が，リファクタリング後は s'_2 が該当する．表 3 より，C(*, *) に該当する文の数はリファクタリングによって 3 から 1 に減っており，それ以外は数に変わりはない．同じ条件分岐先の文が少ないことは，同じ疑惑値を持つ文が少ないことを意味しているため，同じ条件分岐先の文が少ないほど，SBFL 適合性が向上すると考えることができる．

5.2.2 SBFL スコアが低下した例

例としてケース 1, 3 について考察する．ケース 1 で適用した「条件記述の分解」とは，条件文中の記述をメソッドに切り出すリファクタリングである．本手法はメソッド単位で SBFL を行うことを前提としているため，条件式をメソッドではなく変数に切り出すリファクタリングを実験対象とした．条件分岐先ごとに文を分類したところ，表 4 の通り，C(*, *) に該当する文の数が増加していることが確認できた．図 5 より，C(*, *) に該当する文が欠陥箇所であるミュータントのペアにおいては，リファクタリング後のミュータントの方が，*rScore* が低いことが読み取れる．条件式を別の文に切り出したことで同一条件分岐先で実行される文の数が増え，SBFL スコアが低下したと考えられる．

ケース 3 で適用した「重複した条件記述断片の統合」は，if-else の両方で実行される文を if 文の外に切り出すリファクタリングである．条件分岐先ごとに文を分類した結果を表 5 に示す．C(*) に該当する文の数は増加し，C(T), C(F) に該当する文の数は減少しており，ケース 5 と 1 の両方

[*5] ここでは，文 s_1, s_2, s_7 のように必ず実行される文も，条件分岐に影響を受けない 1 つの条件分岐先とみなしている．

	rScore						
ミュータント:	m_1	m_2	m_3	m_4	m_5	m_6	m_7
ミューテーション演算子:	NC	CB	INC	NC	CB	INC	PR
s_1: **if** (0 < n)	**0.67**	**0.50**	0.50	0.50	0.25	0.25	0.75
s_2: n--;	0.33	1.00	**1.00**	0.00	0.00	0.00	0.25
s_3: **else if** (n < 0)	0.00	0.00	0.00	**1.00**	**0.75**	0.75	0.00
s_4: n++;	-	0.00	0.00	0.25	0.75	**1.00**	0.25
s_5: **return** n;	0.67	0.50	0.50	0.50	0.25	0.25	**0.75**

(a) リファクタリング前 (*SBFLScore*=0.81)

	rScore						
ミュータント:	m'_1	m'_2	m'_3	m'_4	m'_5	m'_6	m'_7
ミューテーション演算子:	NC	CB	INC	NC	CB	INC	PR
s'_{1a}: **boolean** f1 = (0 < n);	**0.40**	**0.33**	0.33	0.33	0.17	0.17	0.50
s'_{3a}: **boolean** f2 = (n < 0);	0.40	0.33	0.33	**0.33**	**0.17**	0.17	0.50
s'_{1b}: **if** (f1)	0.40	0.33	0.33	0.33	0.17	0.17	0.50
s'_2 : n--;	0.20	1.00	**1.00**	0.00	0.00	0.00	0.17
s'_{3b}: **else if** (f2)	0.00	0.00	0.00	1.00	0.83	0.83	0.00
s'_4 : n++;	-	0.00	0.00	0.17	0.83	**1.00**	0.17
s'_5 : **return** n;	0.40	0.33	0.33	0.33	0.17	0.17	**0.50**

(b) リファクタリング後 (*SBFLScore*=0.53)

図 5: ケース 1: 条件記述の分解

の事象が起こっている．図 6 よりミュータントのペアを比較すると，リファクタリングによって *rScore* が低下したミュータントのペアの方が，増加したミュータントのペアよりも多い．これは，リファクタリング後に文の数が増加した C(*) に多くの文が該当していることが理由であると考えられる．また，$\langle m_5, m'_{\{5,6\}} \rangle$ と $\langle m_6, m'_{\{5,6\}} \rangle$ のペアは，他のペアと比べてリファクタリング前後の *rScore* の差が大きい．これは，このミュータントの欠陥を含む文が該当する条件分岐先が，リファクタリングによって C(T), C(F) から C(*) に変更となり，それらの分岐先で実行される文の数が 2 から 5 に大きく増えたことが理由であると考えられる．このことから，条件分岐先の文の数が少ない方から多い方へ文が移動したことが，SBFL スコアが低下した原因と考えることができる．逆に，図 6(b) から 6(a) への変換のように，多い方から少ない方へ文の移動を行うことで，SBFL スコアを向上させることができると言える．

5.3 まとめ

実験結果より，同一の条件分岐先で実行される文の数が少ないほど，SBFL 適合性が向上することを確認した．また，これを実現するためのプログラム構造の変換方法とし

表 4: 条件分岐先ごとの文の分類（ケース 1）

(a) リファクタリング前

条件分岐先 C(s_1, s_3)	該当文
C(*, *)	$\{s_1, s_5\}$
C(T, *)	$\{s_2\}$
C(F, *)	$\{s_3\}$
C(F, F)	$\{s_4\}$

(b) リファクタリング後

条件分岐先 C(s'_{1b}, s'_{3b})	該当文
C(*, *)	$\{s'_{1a}, s'_{3a}, s'_{1b}, s'_5\}$
C(T, *)	$\{s'_2\}$
C(F, *)	$\{s'_{3b}\}$
C(F, F)	$\{s'_4\}$

	rScore						
ミュータント:	m_1	m_2	m_3	m_4	m_5	m_6	m_7
ミューテーション演算子:	NC	CB	MA	MA	MA	MA	PR
s_1: **int** result = 0;	0.57	0.29	0.29	0.29	0.29	0.29	0.57
s_2: **int** tmp = 0;	0.57	0.29	0.29	0.29	0.29	0.29	0.57
s_3: **if** (x > 0) {	**0.57**	**0.29**	0.29	0.29	0.29	0.29	0.57
s_4: tmp = y * 2;	0.29	0.86	**0.86**	0.00	0.86	0.00	0.00
s_5: result = y + tmp;	0.29	0.86	0.86	0.00	**0.86**	0.00	0.00
} **else** {							
s_6: tmp = y * 3;	0.00	0.00	0.00	**0.86**	0.00	0.86	0.29
s_7: resut = y + tmp;	0.00	0.00	0.00	0.86	0.00	**0.86**	0.29
}							
s_8: **return** result;	0.57	0.29	0.29	0.29	0.29	0.29	**0.57**

(a) リファクタリング前 (*SBFLScore*=0.69)

	rScore					
ミュータント:	m'_1	m'_2	m'_3	m'_4	$m'_{\{5,6\}}$	m'_7
ミューテーション演算子:	NC	CB	MA	MA	MA	PR
s'_1 : **int** result = 0;	0.33	0.17	0.17	0.17	0.33	0.33
s'_2 : **int** tmp = 0;	0.33	0.17	0.17	0.17	0.33	0.33
s'_3 : **if** (x > 0)	**0.33**	**0.17**	0.17	0.17	0.33	0.33
s'_4 : tmp = y * 2;	0.17	1.00	**1.00**	0.00	0.00	0.00
else						
s'_6 : tmp = y * 3;	0.00	0.00	0.00	**1.00**	0.17	0.17
$s'_{\{5,7\}}$: result = y + tmp;	0.33	0.17	0.17	0.17	**0.33**	0.33
s'_8 : **return** result;	0.33	0.17	0.17	0.17	0.33	**0.33**

(b) リファクタリング後 (*SBFLScore*=0.53)

図 6: ケース 3: 重複した条件記述断片の統合

て，以下の手段が有効であると考えられる．

- early return を用いて条件分岐を早く終了させる
- 同一の条件分岐先の文の数が多い方から少ない方へ文を移動する

6. 妥当性への脅威

本論文では表 2 に記載の 5 種類のリファクタリングを対象とした．リファクタリングパターンは数多く存在するため，他のパターンで検証することで，新たな傾向を発見したり，今回発見した傾向を否定する例が現れたりする可能性がある．また，本論文では表 1 に記載の 11 種類のミューテーション演算子を提案手法の実装に用いて，これらのミューテーション演算子が適用されるよう考慮して実験対象プログラムを作成した．より多くのミューテーション演算子が適用できれば，異なる結果が得られる可能性がある．

本実験の対象としたプログラムは，リファクタリングパターンを再現するために手作業で作成した非常に簡素なプ

表 5: 条件分岐先ごとの文の分類（ケース 3）

(a) リファクタリング前

条件分岐先 C(s_3)	該当文
C(*)	$\{s_1, s_2, s_3, s_8\}$
C(T)	$\{s_4, s_5\}$
C(F)	$\{s_6, s_7\}$

(b) リファクタリング後

条件分岐先 C(s'_3)	該当文
C(*)	$\{s'_1, s'_2, s'_3, s'_{\{5,7\}}, s'_8\}$
C(T)	$\{s'_4\}$
C(F)	$\{s'_6\}$

```
s1: if (x > 0)
s2:   return 10;
s3: if (y < 0)
s4:   return 10;
s5: if (z == 0)
s6:   return 10;
s7: return 20;
```

(a) リファクタリング前

```
s'{1,3,5} : if( (x>0)
                ‖ (y<0)
                ‖ (z==0))
s'{2,4,6} :   return 10;
s'7       : return 20;
```

(b) リファクタリング後

図 7: ケース 2：条件式の統合

```
s1: int r = 0;
s2: boolean found = false;
s3: for (int i : array) {
s4:   if (found != true) {
s5:     if (i == 0)
s6:       found = true;
s7:     result = result + i;
      }}
s8: return result;
```

(a) リファクタリング前

```
s'1: int r = 0;
s'3: for (int i : array) {
s'5:     if (i == 0)
s'9:       break;
s'7:     result = result + i;
      }
s'8: return result;
```

(b) リファクタリング後

図 8: ケース 4：制御フラグの削除

ログラムであった．そのため，実在するプログラムや規模の大きいプログラムを対象とした場合，異なる結果が得られたり，提案手法に対する課題を発見したりする可能性がある．また，テストスイートは，実験対象のプログラムと全てのミュータントの C2 カバレッジを満たすよう手作業で作成した．このため，あるミュータントへのテスト実行時に全く同じ実行経路を取るような無駄なテストケースが多く存在してしまっている．テストスイートは SBFL 結果に影響を及ぼす要素であるため，テストスイートの作成方法次第で，今回と異なる結果が得られる可能性がある．

7. おわりに

本論文では，プログラム自体が SBFL にどの程度適しているかという特性を持っていると考え，その特性を SBFL 適合性と定義した．また，ミューテーションテストを活用した SBFL 適合性の計測手法を提案した．リファクタリングを題材に，プログラム構造の違いによる SBFL 適合性の違いを確認した結果，同じ機能，同じテストスイートであっても，プログラム構造の違いにより SBFL 適合性が変化することを確認した．また，SBFL 適合性を向上させるプログラム構造の特徴を発見した．

今後の課題として，実在するプログラムに対して同様の検証を行うことで，結果の一般化を図る必要がある．また，元のプログラムを SBFL 適合性の高いプログラムに変換する提案を今後は検討する．SBFL を実行する前に，SBFL 適合性が高いプログラム構造に変換することで，欠陥限局の精度を向上させることができる可能性がある．

謝辞 本論文の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222），基盤研究（B）（課題番号：20H04166）の助成を得て行われた．

参考文献

[1] Hailpern, B. and Santhanam, P.: Software debugging, testing, and verification, *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12 (2002).
[2] Britton, T., Jeng, L., Carver, G. and Cheak, P.: Reversible Debugging Software “Quantify the time and cost saved using reversible debuggers” (2013).
[3] Wong, W. E., Gao, R., Li, Y., Abreu, R. and Wotawa, F.: A survey on software fault localization, *IEEE TSE*, Vol. 42, No. 8, pp. 707–740 (2016).
[4] Jia, Y. and Harman, M.: An Analysis and Survey of the Development of Mutation Testing, *IEEE TSE*, Vol. 37, No. 5, pp. 649–678 (2011).
[5] Jones, J. A., Harrold, M. J. and Stasko, J.: Visualization of test information to assist fault localization, *Proc. ICSE*, pp. 467–477 (2002).
[6] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A. and Brewer, E.: Pinpoint: Problem determination in large, dynamic Internet services, *Proc. DSN*, pp. 595–604 (2002).
[7] Dallmeier, V., Lindig, C. and Zeller, A.: Lightweight Defect Localization for Java, *Proc. ECOOP*, pp. 528–550 (2005).
[8] da Silva Meyer, A., Garcia, A. A. F., de Souza, A. P. and de Souza Jr., C. L.: Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L), *Genetics and Molecular Biology*, Vol. 27, No. 1, pp. 83–91 (2004).
[9] Abreu, R., Zoeteweij, P. and van Gemund, A. J. C.: On the Accuracy of Spectrum-based Fault Localization, *Proc. TAIC PART*, pp. 89–98 (2007).
[10] Wang, T. and Roychoudhury, A.: Automated Path Generation for Software Fault Localization, *Proc. ASE*, pp. 347–351 (2005).
[11] Lu, H., Gao, R., Huang, S. and Wong, W. E.: Spectrum-Base Fault Localization by Exploiting the Failure Path, *Proc. ASE*, pp. 252–257 (2016).
[12] Ma, Y.-S., Offutt, J. and Kwon, Y.-R.: MuJava: A Mutation System for Java, *Proc. ICSE*, pp. 827–830 (2006).
[13] Just, R.: The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java, *Proc. ISSTA*, pp. 433–436 (2014).
[14] Coles, H., Laurent, T., Henard, C., Papadakis, M. and Ventresque, A.: PIT: A Practical Mutation Testing Tool for Java (Demo), *Proc. ISSTA*, pp. 449–452 (2016).
[15] Ramler, R., Wetzlmaier, T. and Klammer, C.: An Empirical Study on the Application of Mutation Testing for a Safety-Critical Industrial Software System, *Proc. SAC*, pp. 1401–1408 (2017).
[16] Moon, S., Kim, Y., Kim, M. and Yoo, S.: Ask the Mutants: Mutating Faulty Programs for Fault Localization, *Proc. ICST*, pp. 153–162 (2014).
[17] Papadakis, M. and Le Traon, Y.: Metallaxis-FL: Mutation-Based Fault Localization, *Journal of STVR*, Vol. 25, No. 5–7, pp. 605–628 (2015).
[18] Boswell, D. and Foucher, T.: *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*, O’Reilly Media (2011).
[19] Coles, H.: PIT, , available from ⟨https://pitest.org/⟩ (accessed 2020).
[20] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).