Vol.2015-SE-190 No.5
2015/12/15

影響波及を考慮したフレームワークアプリケーションの障害位置特定手法

森岡 友樹[1,a)]　新田 直也[1,b)]

概要：ソフトウェア開発において, デバッグは最も時間を要する作業の 1 つである. そこで近年, デバッグ作業の効率化を図るため障害の混入位置の特定を支援する障害位置特定技術が研究されている. 障害の混入位置と故障の発生個所の対応関係が比較的単純な障害においては自動化された障害位置特定手法も提案されている. しかしながら, 単一の障害が広範囲に影響するような複雑な障害を対象とした支援手法は今のところ見当たらない. そこで本研究では, 対象をフレームワークアプリケーションに限定し, 実際にプログラムに施した修正の前後でのテスト結果を比較することによって, 障害位置を系統的に特定する手法の提案を行う. 本稿では, オープンソースフレームワークを用いた事例において本手法が有効であることを確認したので報告する.

キーワード：影響波及, 障害位置特定, フレームワークアプリケーション

1. はじめに

ソフトウェア開発において, デバッグは最も時間を要する作業の 1 つである. デバッグ作業には障害位置の特定, 障害の除去などの作業が含まれるが, とりわけ障害位置の特定作業はソフトウェアの規模が大きくなればなるほど, また対象ソフトウェアに対する知識が不足していればいるほど作業者への負荷が増大する. そこで近年, 障害が混入したソフトウェアの振る舞いやソースコードなどの情報を元に, 障害が混入している可能性の高いソースコード上の箇所を推定する**障害位置特定技術**が研究されている. たとえば, 文献 [1] では, 障害を含んだソフトウェアを複数のテストケースに対して実行したときの, 各コンポーネントの実行の有無および故障の発生の有無の情報から障害が混入しているコンポーネントを推測する**スペクトラム**に基づく自動障害位置特定手法が提案されている. この手法では, 各テストケースに対する故障の発生の有無のパターンとコンポーネントの実行の有無のパターンの類似度が高いコンポーネントほど障害の混入可能性が高いとみなされる. しかしながら, この手法で暗黙に前提されている故障の発生メカニズムは, 障害を含んだコンポーネントの実行と故障の発生が直結するという比較的単純なもので, たとえば障害を含んだコンポーネントを実行してもすぐには故障が発生せず, その影響が他のコンポーネントの実行に及ぶことによって初めて故障として顕在化するといったより複雑な故障のメカニズムについては考慮されていない.

そこで本研究では, 影響波及を考慮したより複雑な故障の発生メカニズムを導入することによって, 文献 [1] の自動障害位置特定手法の拡張を行う. 具体的には, ある障害を除去しようと実際に修正を行った後に別の不具合が顕在化するような状況を想定し, 修正前のプログラムを各テストケースに対して実行した際の結果と, 修正後のプログラムに対する同様の結果を比較することによって, より正確な障害位置の特定を目指す. ただし, 実用規模プログラムにおけるコンポーネントの膨大な組み合わせを考慮して提案手法を構築することは現実的に困難であることが予想されるため, 対象をフレームワークアプリケーションに限定し, その中のアプリケーション部分のコンポーネントのみを修正対象とすることでコンポーネントの組み合わせを削減することを考える. これは一般に, フレームワーク内部の実装は比較的安定していると考えられること, フレームワーク内部の実装の修正は影響範囲の大きさから避けられることが多いことなどを考えたときに, 現実的な制約であると考えられる.

本稿では, オープンソースフレームワークにおけるアプリケーションの不具合を対象に本手法を適用し, 影響波及を考慮に入れつつ正しい修正箇所を提示できることを確認

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m1424009@s.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

した. 今後, 修正前後のプログラムに対して動的解析を行うことによって, 障害位置の推定手続きを自動化し, 障害位置特定ツールを開発することを予定している.

2. 障害位置特定

テスト作業では, テスト対象となるプログラムに対する操作手順と期待される振る舞いを記述した複数のテストケースを実行することでプログラムの故障の有無を調査することが多い. 故障が発見された場合はその原因となる障害がソースコード上で混入している位置を特定し不具合の除去が行われる. 障害位置特定技術は, この障害の位置特定作業を支援するために研究されている技術である. 障害位置特定のアプローチとしては, スペクトラムに基づいた自動障害位置特定手法 (文献 [1] 参照. 以下, 既存手法とよぶ) が挙げられる. この手法では, プログラムが複数のコンポーネント (クラス, メソッドなど) によって構成されており, テスト作業に当たって複数のテストケースが用意されていることを仮定する. ここで, 解析対象のプログラムが M 個のコンポーネントで構成されており, そのプログラムに対して, N 個のテストケースが用意されているとする. 各テストケース $T_n(1 \leq n \leq N)$ を実行した時の各コンポーネント $C_m(1 \leq m \leq M)$ の活動状態を T_n のスペクトラム (文献 [2] 参照) という. 本稿では, 活動状態としてコンポーネントのそのテストケースにおける実行の有無のみを対象とし, $N \times M$ **アクティビティ行列** A の各成分 a_{nm} で表す.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NM} \end{bmatrix} \tag{1}$$

これらの情報は, 各テストケースを実行した際に収集され, 式 (1) のように表現される. T_n 実行時に C_m の実行が確認された場合, $a_{nm} = 1$ となり, それ以外の場合は $a_{nm} = 0$ となる. 同時に, テストケースが成功もしくは失敗したという情報を**エラーベクトル** e で表現する. ただし, e の各成分 $e_n(1 \leq n \leq N)$ はテストケース T_n を実行した際にそのテストが成功した場合 $e_n = 0$ となり, 失敗した場合は $e_n = 1$ となる. 既存手法では (A, e) の組のみが入力となる. 既存手法では, 各コンポーネント C_m の活動情報 a_{*m} とエラーベクトル e の間の統計的類似性を計算し C_m における障害の混入可能性を推定する. この統計的類似性は**類似度** (文献 [3] 参照) によって定量化され, 以下のように算出される. **ヒットカウンタ** $n_{pq}(C_m)$ は, a_{*m} と e_* の各成分の値がそれぞれ p と q $(p, q \in \{0, 1\})$ であるテストケースの数を示し,

$$n_{pq}(C_m) = \left|\{n \mid a_{nm} = p \wedge e_n = q\}\right|$$

と定義される. これら 4 通りのヒットカウンタの値から, 落合係数 $s_{\mathrm{O}}(C_m)$(文献 [4] 参照) を以下の式で求める.

$$s_{\mathrm{O}}(C_m) = \frac{n_{11}(C_m)}{\sqrt{(n_{11}(C_m) + n_{10}(C_m)) \cdot (n_{11}(C_m) + n_{01}(C_m))}}$$

表 1 に, 既存手法における入力 (A, e) の例と, そこから求められたヒットカウンタ及び類似度の計算結果の一例を示す.

表 1 既存手法の入力及び出力一例

	C_1	C_2	e
T_1	1	0	1
T_2	1	1	1
$n_{01}(C_{\mathrm{m}})$	0	1	
$n_{10}(C_{\mathrm{m}})$	0	0	
$n_{11}(C_{\mathrm{m}})$	2	1	
$s_{\mathrm{O}}(C_m)$	1.0	0.7	

表 1 において, テスト対象のプログラムに何らかの故障 (以下, エラーと呼ぶ)e が存在していることがわかる. この場合, a_{*1} の類似度が最も高いため C_1 が修正を行うべき**障害コンポーネント**であると推定される. 既存手法ではこの診断結果に基づき C_1 を修正することで障害の除去を行う.

しかしながら, 既存手法では障害を修正することによって新たにエラーが混入するような場合についての考察がなされていない. 例えば, 先ほどの修正で取り除いたエラー e とは異なる新たなエラー e' が発生するような状況において, 修正を元に戻したうえで改めて C_1 を修正すべきか, 修正後のプログラムに対し再度障害位置特定を行い推定された別のコンポーネント C' を修正すべきなのかが不明確である. このように, プログラムのある部分の修正がプログラムの他の部分に影響を与えることで新たに障害が混入される状況も少なくない.

そこで本研究では, 既存手法を拡張することで, 前述のような状況において修正を続行するべきか判断すると同時に, 修正すべきコンポーネントを特定できる自動障害位置特定手法の提案 (以下, 提案手法と呼ぶ) を行う.

3. フレームワークアプリケーションの障害位置特定手法

本節では, 提案手法の説明を行う.

まず, 提案手法が対象とするプログラムをフレームワークアプリケーションとし, 修正候補をそのアプリケーション固有部分に限定する. その理由は,

(1) 一般にフレームワーク内部の実装は比較的安定しているため, 修正可能な箇所をアプリケーション固有部分に限定することによって, 修正候補の探索空間を小さくできること,
(2) フレームワーク内部の実装の修正は影響範囲の大きさから実際に避けられる場合が多いこと,

の 2 点である. 手法の詳細を説明する前に, フレームワークアプリケーションについて説明する.

3.1 アプリケーションフレームワーク

アプリケーションフレームワーク (文献 [5] 参照, 以下フレームワークと略する) とは, 特定のドメイン内で繰り返し出現するオブジェクト間の協調やアプリケーション全体の制御構造などを抽象化した再利用可能なクラス群である. これらのクラスにはアプリケーション側で変更可能なホットスポット (文献 [6] 参照) が抽象メソッドとして用意されており, フレームワークアプリケーションの開発者は, このような抽象メソッドをアプリケーション側で作成した子クラスでオーバーライドすることによってアプリケーション固有の振る舞いを実装する. アプリケーション固有の振る舞いは, アプリケーション側で実装したメソッドが実行時にフレームワーク側から呼び出されることによって実行される. このような制御の仕組みを一般に**制御の反転**と呼ぶ. 例えば, イベント駆動型フレームワークにおける**イベントハンドラ**は一般に制御の反転によって呼び出される. 本研究では修正候補をフレームワークアプリケーションのアプリケーション固有部分に限定しており, コンポーネントは制御の反転で呼び出されるメソッド及びその制御の反転を設定するコードに相当する.

3.2 手法の全体の流れ

本節では提案手法の流れについて説明する. 対象となるフレームワークアプリケーションは, アプリケーション固有部分に M 個のコンポーネントを持つものとし, あらかじめ用意された N 個のテストケースの実行時にエラー e が発生したとする. ここで, e の原因となる障害コンポーネント C_k を修正した結果, 回帰テストにおいて新たにエラー e' が発生したという場合を考える. なお文献 [1] にならい, 対象となるプログラムに含まれている障害コンポーネントは 1 つのみと仮定する.

既存手法では, 障害コンポーネントが実行されることでエラーが発生し, 実行されなかった場合はエラーが発生しないという単純なエラー発生過程モデルを前提としていた. しかし実際には障害コンポーネントを実行した直後ではなく, 別のコンポーネントが実行されたときにエラーが発生する場合が存在する. そこで提案手法では, 以下の 2 種類のエラー発生過程を扱えるように, 既存手法のエラー発生過程モデルを拡張する.

- 障害コンポーネントを実行することにより発生するエラー (**直接型**)
- 障害コンポーネント実行時にメモリ上に不正な状態が保持され, その状態が別のコンポーネントの実行時に参照される (**影響波及**) ことにより発生するエラー (**副作用型**)

また, 上記の各エラー発生過程モデルを仮定したときの, 関連するコンポーネントの活動状態に応じて変化するエラーの発生可能性を表す**エラー発生可能性**を定義する.

- $\tilde{e}_{\mathrm{D}}(C_i)$: C_i を原因とする直接型エラーの発生可能性
- $\tilde{e}_{\mathrm{S}}(C_i, C_j)$: C_i を原因とし, C_j 実行において発生する副作用型エラーの発生可能性

副作用型エラー発生可能性 $\tilde{e}_{\mathrm{S}}(C_i, C_j)$ を考慮するにあたり, 各テストケースの実行順の情報が必要となる. 加えて, e' が影響波及により発生したエラーであるかを推定するために, 修正前後それぞれのプログラムに対してテストケースの実行結果を比較する必要がある. そこで, これらの情報を新たに入力として与えられるようにアクティビティ行列を拡張する. まず, C_k を修正した後のコンポーネント ${C_k}'$ を加えた $M+1$ 個のコンポーネントに対してテストケースを実行すると仮定する. さらに, 修正前のプログラムに対して実行したテストケース T_n に対して, 修正後に実行したテストケースを ${T_n}'$ とする. このとき, 修正前後それぞれのプログラムに対してテストケースを実行した際のアクティビティ行列を連結したものを $(2N) \times (M+1)$ **拡張アクティビティ行列** A' とし, A' の各成分を ${a'}_{ik}$ で表す. 表 2 に, 拡張アクティビティ行列の一例を示す.

表 2 拡張アクティビティ行列

	C_1	$C_1{}'$	C_2
T_1	1	0	0
T_2	1	0	1
$T_1{}'$	0	1	0
$T_2{}'$	0	1	1

e	e'
1	0
1	0
0	0
0	1

ここで, 提案手法における入力を以下に示す.

- 拡張アクティビティ行列 A'
- 各テストケースにおけるコンポーネントの実行順
- 修正前のプログラムで発生したエラーに対するエラーベクトル e
- 修正後のプログラムで発生したエラーに対するエラーベクトル e'

これらの入力を基に, **エラー発生可能性指標** $\tilde{E}$ を求める. エラー発生可能性指標 $\tilde{E}$ は, 入力された情報を基にあらゆるエラー発生過程モデルを仮定したときのエラーの発生可能性を 1 つの表にしたもので, 以下のように定義される.

$$\tilde{E} = \{\tilde{e}_{\mathrm{D}}(C_i) \mid 1 \leq i \leq M\} \cup \{\tilde{e}_{\mathrm{D}}(C_k)'\} \cup \{\tilde{e}_{\mathrm{S}}(C_i, C_j) \mid \exists_T . C_i \overset{T}{\prec} C_j\}$$

ここで, $C_i \overset{T}{\prec} C_j$ はテストケース T の実行において C_j より C_i が先に実行されたことを表す. また, エラー発生可能性指標の各成分は以下のように算出される.

$\tilde{e}_{\mathrm{D}}(C_i)(T_k) = {a'}_{ik}$

($1 \leq k \leq 2N$, ただし $1 \leq k \leq N$ に対して $T_{k+N} = {T_k}'$)

$$\tilde{e}_{\mathrm{S}}(C_i, C_j)(T_k) = \begin{cases} 0 & \text{if } C_i \overset{T_k}{\not\prec} C_j \\ 1 & \text{if } C_i \overset{T_k}{\prec} C_j \end{cases}$$

ここで $\tilde{E}$ の算出結果に 2 つのエラーベクトルを併記したものを表 3 に示す.

$\tilde{E}$ を基に, 修正前後それぞれでエラー発生可能性とエラーベクトルの類似度を比較する. ここで, $\tilde{e}_{\mathrm{D}}(C_i)$ とエラーベクトル e の類似度を $s_{\mathrm{O}}(\tilde{e}_{\mathrm{D}}(C_i), e)$, $\tilde{e}_{\mathrm{S}}(C_i, C_j)$ とエラーベクトル e の類似度を $s_{\mathrm{O}}(\tilde{e}_{\mathrm{S}}(C_i, C_j), e)$ と表し, 類似度計算は既存手法と同様に落合係数を用いる. 本手法では修正後の類似度に着目する. 類似度が最大となるエラー発生可能性が $s_{\mathrm{O}}(\tilde{e}_{\mathrm{D}}(C_i), e')$, $s_{\mathrm{O}}(\tilde{e}_{\mathrm{S}}(C_i, C_j), e')$ いずれの場合においても提案手法で推定される障害位置候補は C_i となる.

4. 事例研究

提案手法が有効であるかどうかを確かめるため, 実際のフレームワークアプリケーションを用いた事例研究を行った. jMonkeyEngine[7](以下 jME とする) というオープンソースフレームワークを対象として, 本手法を適用しすべて手動でサンプルアプリケーションの修正を試みた.

4.1 jMonkeyEngine

jME は, Java で利用することが可能な 3D ゲームエンジンである. jME にはいくつかのサンプルアプリケーションが付属しており, 事例研究では, そのうちの 1 つである Lesson8 (図 1 参照) を対象アプリケーションとする. このアプリケーションのテストケースが存在しなかったため, テストケースは我々が作成した (図 2 参照). また, 本稿における事例研究はすべて手作業で行うため, 対象コンポーネント群は本事例に関わると思われるもののみ抜粋する.

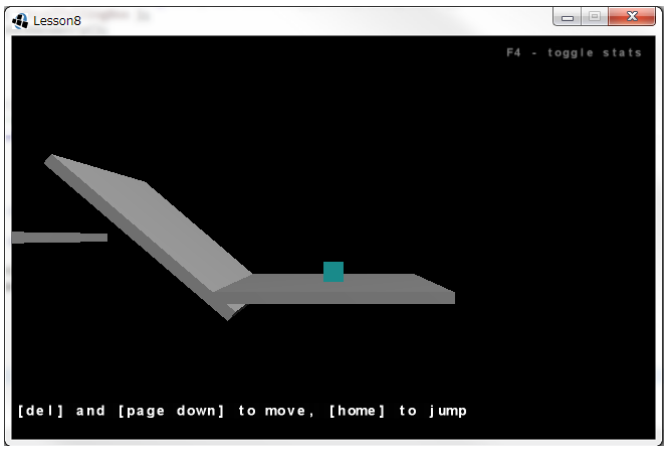

図 1 Lesson8 の実行画面

テストケース T_1 ではイベントハンドラ `MoveAction.performAction()` (以後, C_1 とする) が, テストケース T_2 ではさらに別のイベントハンドラ `Lesson8$2.performAction()` (以後, C_2 とする) が呼び出されている. また, いずれのテストケースにおいても期待した振る舞いを満たさなかった. この時点では影響波及を確認できないため提案手法は既存手法と同等である. そこで既存手法を適用すると表 1 に示すような結果が得られ C_1 が障害位置として推定される. そこで, 以下の図 3 で表される修正を行う. 図 3 において, 破線で囲まれた部分が実際に修正を加えたコードとなる.

```
input.addAction( new MoveAction( new Vector3f( -5, 0, 0 ) ),
          InputHandler.DEVICE_KEYBOARD,
          KeyInput.KEY_DELETE,
          InputHandler.AXIS_NONE,
          true );
input.addAction( new MoveAction( new Vector3f( 5, 0, 0 ) ),
          InputHandler.DEVICE_KEYBOARD,
          KeyInput.KEY_PGDN,
          InputHandler.AXIS_NONE,
          true );
    .
    .
    .
private class MoveAction extends InputAction {
        .
        .
        .
    public void performAction( InputActionEvent evt ) {
        if(playerOnFloor){
            if(evt.getTriggerPressed()){
                player.setLinearVelocity(direction);
            }else{
                player.setLinearVelocity(ZERO);
            }
        }
    }
}
```

図 3 修正 1 回目におけるコード修正箇所

修正後に回帰テストとして同じテストケースを実行した結果, 新たに異なるエラーの発生が確認された. 修正前のエラーを e, 修正後のエラーを e' とする. ここで, 修正の前後でテストケースを実行し, 各コンポーネントの活動状態を記録した拡張アクティビティ行列およびエラーベクトルを表 2 に示す. また, これらの情報を基に算出したエラー発生可能性指標を表 3 に示す. さらに, この指標を基にした類似度の算出結果を表 4 に示す. 表 4 から, $s_{\mathrm{O}}(\tilde{e}_{\mathrm{S}}(C_1{}', C_2), e)$ が最も高い類似度を示したことがわかる. これは C_1 の修正が C_2 の実行にへ影響波及を起こしていたためと推定できる. よって, この場合は修正前のコードを参考にし, C_2 の実行への影響がないように C_1 を再度修正する方針が推奨される. 図 4 は実際に影響波及を考慮した修正を行った後のソースコードの一部であり, 破線で囲まれた部分が実際に修正を加えたコードとなる. この修正の結果, エラーが発生することなく, テストケースをすべて通すことに成功した.

5. 考察と今後の課題

前節で行った jME を用いた事例研究において, 本手法を適用することによって全てのテストケースを通すことに成功した. このことから本手法による障害位置特定が有効で

表 3　エラー発生可能性指標

	C_1	C_1'	C_2	$\tilde{e}_{\mathrm{D}}(C_1)$	$\tilde{e}_{\mathrm{D}}(C_1')$	$\tilde{e}_{\mathrm{S}}(C_1, C_2)$	$\tilde{e}_{\mathrm{S}}(C_1', C_2)$	$\tilde{e}_{\mathrm{D}}(C_2)$	e	e'
T_1	1	0	0	1	0	0	0	0	1	0
T_2	1	0	1	1	0	1	0	1	1	0
T_1'	0	1	0	0	1	0	0	0	0	0
T_2'	0	1	1	0	1	0	1	1	0	1

[テストケース 1 (T_1)]
期待される振る舞い：キャラクターが水平な地面の上にいるとき，
　右移動 (左移動) ボタンを押すとキャラクターが地面の上を右 (左) に瞬時に移動する．

[テストケース 2 (T_2)]
期待される振る舞い：キャラクターが水平な地面の上にいるとき，
　右移動 (左移動) ボタンを押すとキャラクターが地面の上を右 (左) に瞬時に移動し，
　斜面にさしかかると，キャラクターが斜面の方向に移動を続ける．

図 2　jMonkeyEngine のテストケース

```
private class MoveAction extends InputAction {
    .
    .
    .
    public void performAction(InputActionEvent evt){
        if(playerOnFloor){
            if(evt.getTriggerPressed()){
                Vector3f vel =
                    normal.cross(new Vector3f(0.0f, 0.0f, 1.0f));
                float scale = direction.length();
                if(direction.getX() > 0){
                    scale = -scale;
                }
                vel.scaleAdd(scale, ZERO);
                player.setLinearVelocity(vel);
            } else {
                player.setLinearVelocity(ZERO);
            }
        }
    }
}
```

図 4　修正 2 回目におけるコード修正箇所

あったと考えられる．しかし，これらの事例において，手動での確認であることに加え，関連性があると考えられるコンポーネントのみを抽出して本手法の検証を行ったため，動的解析等を用いて自動でコンポーネントを抽出し，対象コンポーネントの数を絞り込まない状況であっても障害位置特定が成功するかの検証を行う必要がある．

また，対象コンポーネントの数に対してテストケース数が不十分な状態の場合，類似度による判定だけでは障害位置を正しく推定できないことが考えられる．そのため，現実のソフトウェア開発において障害位置を絞り込むのに十分な数のテストケースが一般に存在しているかについて検証していきたい．

今後の課題として，本手法のさらなる効率化を図るため，以下に挙げる本手法の一部の作業を自動化または半自動化することを検討している．

- テストケース実行時におけるコンポーネント活動情報の取得
- エラー発生可能性指標の生成及び類似度計算

また，多くの事例への適用を行い，本手法の有効性のより厳密な検証を行っていきたい．フレームワークに不慣れな利用者にとって影響波及がある場合の障害位置の特定作業は時間を要するものであり，一般にフレームワーク及びアプリケーションの規模が大きくなるほどその作業時間は増大していく．よって，動的解析技術を用いることで本手法を自動化もしくは半自動化することにより，これらの作業を効率化することを考えている．

6. 関連研究

障害位置特定は使用している技術に応じて大きく 2 つに分類される．一つは，実行トレースにおいて，実行されたプログラムの要素と対応するスペクトラムに基づいて障害位置を特定する手法である．もう一つはプログラムの利用者によって提供される不具合報告に記述されているソースコード中の名称やコメントとの関連性から障害位置を特定する情報検索を元にした手法である．スペクトラムに基づいた手法としては，文献 [8] では，プログラム部品の不審度を順位付けする式に Tarantula 係数を利用する手法が提案されている．文献 [9] では別の係数である落合係数を用いての障害位置特定が試みられており，文献 [10] では複数の障害が混入している場合についての障害位置特定手法が提案されている．文献 [11] では，複数の関連指標を調査し，障害位置特定に有効な指標について考察がなされている．本研究もスペクトラムに基づいた手法の一種である．近年では，それまでの経験則に基づいた計算式ではなく，学習型プログラミング (文献 [12] 参照) や遺伝的プログラミング (文献 [13] 参照) によってより正確に障害位置特定を行えるような類似度計算式を構築する研究も行われている．スペクトラムに基づいた手法の基本的な方針としては，失敗時の実行トレースにおいて頻繁に実行されたプログラム要素の順位をリスト形式で出力するが，多くの場合，障害を含むプログラムのブロックあるいは行単位で出力される．しか

表 4　事例研究における類似度算出結果

	$s_{\mathrm{O}}(\tilde{e}_{\mathrm{D}}(C_1), e)$	$s_{\mathrm{O}}(\tilde{e}_{\mathrm{D}}(C_1'), e)$	$s_{\mathrm{O}}(\tilde{e}_{\mathrm{S}}(C_1, C_2), e)$	$s_{\mathrm{O}}(\tilde{e}_{\mathrm{S}}(C_1', C_2), e)$	$s_{\mathrm{O}}(\tilde{e}_{\mathrm{D}}(C_2), e)$
e	1.0	0.0	0.5	0.0	0.7
e'	0.0	0.7	0.0	1.0	0.7

し, 障害を含むソースコードのうちの行単位での障害位置特定は扱いやすい反面, 障害が複数行に散在していることもあるため必要な修正箇所の情報が不足する可能性も考えられる.

情報検索手法では, 不具合報告とメソッドやファイルのようなプログラム要素との類似性を計算するために情報検索技術が用いられ, 潜在的意味分析による機能特定手法 (文献 [14] 参照) などが挙げられる. この手法では障害に関する記述に最も関連のあるプログラム要素の順位リストを出力するが, 多くの場合は障害を含むソースコードのファイル単位で出力される. ただし, 情報量が多く実際に障害を含んでいる数行のソースコードを見つけるために多くのコードを読み飛ばす作業が必要とされる.

また, これら 2 つの手法を組み合わせて障害位置特定を試みる研究もおこなわれており, 文献 [15] では不具合報告と実行トレースの両方を入力として, 障害位置候補をメソッド単位で出力する手法が提案されている.

7. おわりに

影響波及を考慮した障害位置特定を支援する手法の提案を行った. 本手法は, 対象をフレームワークアプリケーションに限定することによって, 複雑なエラー発生過程を系統化し, 無駄な修正作業や回帰テストの低減を目指したものである. オープンソースフレームワークにおける不具合を対象に本手法の適用を行ったところ, 不必要な作業を発生することなく, 正常に障害位置特定が可能であることを確認した. 今後, 本手法の一部の作業を自動化もしくは半自動化するツールを開発して, 手法全体の効率化を図っていく予定である.

参考文献

[1] Abreu Rui, Peter Zoeteweij, and Arjan JC Van Gemund: On the accuracy of spectrum-based fault localization, *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 89–98 (2007).

[2] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi: An empirical investigation of program spectra, In *Proceedings of International Workshop on Program Analysis for Software Tools and Engineering (PASTE' 98)* (1998).

[3] A. da Silva Meyer, A. A. Franco Farcia, and A. Pereira de Souza: 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).

[4] Ochiai, Akira: Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions, *Bull. Jpn. Soc. Sci. Fish* , Vol. 22, No. 9, pp. 526–530 (1957).

[5] Mohamed E. Fayad, Ralph E. Johnson and Douglas C. Schmidt. *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons (1999).

[6] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*. Addison Wesley (1995).

[7] jMonkeyEngine, 入手先 ⟨http://jmonkeyengine.org⟩ (参照 2015–11–16).

[8] J. A. Jones and M. J. Harrold: Empirical evaluation of the tarantula automatic fault-localization technique, In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7–11, 2005, Long Beach, CA, USA*, pp. 273–282 (2005).

[9] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund: A practical evaluation of spectrum-based fault localization, *Journal of Systems and Software* (2009).

[10] Abreu Rui, Peter Zoeteweij, and Arjan JC Van Gemund: Spectrum-based multiple fault localization, *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pp. 88–99 (2009).

[11] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi: Extended comprehensive study of association measures for fault localization, *Journal of Software: Evolution and Process*, Vol. 26, No. 2, pp. 172–219 (2014).

[12] S. Yoo: Evolving human competitive spectra-based fault localisation techniques, In *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28–30, 2012. Proceedings*, pp. 244–258 (2012).

[13] J. Xuan and M. Monperrus: Learning to combine multiple ranking metrics for fault localization, In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 – October 3, 2014*, pp. 191–200 (2014).

[14] A. Marcus and J. I. Maletic: Recovering documentation-to-source-code traceability links using latent semantic indexin,. In *Proceedings of the 25th International Conference on Software Engineering, May 3–10, 2003, Portland, Oregon, USA*, pp. 125–137 (2003).

[15] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo: Information retrieval and spectrum based bug localization: better together, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, August 30 – September 4, 2015, Bergamo, Italy*, pp. 579–590 (2015).