Vol.2021-SE-209 No.10
2021/11/25

SBFL 手法の適用によりサーバサイドアプリケーションの障害原因を特定する手法

謝 佳智[1]　名倉 正剛[2]　高田 眞吾[1]

概要：近年のソフトウェアは複数のコンポーネントから成り，その構成は複雑化してきている．そのため，障害が発生した際にその原因を特定することが非常に困難となっている．実際に障害が生じた際には，蓄積されたログを参考に障害原因を特定するが，大規模で複雑なシステムでは非常に膨大な量のログが生成される．この膨大なログの中から注目するべきログメッセージを見つけ出すためには多大な労力を要さなければならない．特に連続した稼働が求められるようなシステムでは障害原因の特定は速やかに行わなければならないため，この作業を効率化する必要がある．本研究では，サーバサイドアプリケーションを対象に，プログラムコード中の障害原因を特定する技術である Spectrum-Based Fault Localization 手法をログに適用して，障害原因に関連するログメッセージを特定する手法を提案する．提案手法は，サーバサイドアプリケーションを構成する複数のコンポーネントから生成されるログファイルの中から，必要なログメッセージを抽出・分析することで速やかな障害原因特定の実現をサポートする．

キーワード：ログ解析，フォルトローカライゼーション，LAMP，障害原因分析

1. はじめに

近年のソフトウェアは，大規模化により障害が生じた際にその原因を特定することが非常に困難となっている．特に，複数のコンポーネントで構成されるシステムではその作業がより複雑になる．一般的に普及しているサーバサイド Web アプリケーションを実現するために利用するソフトウェアスタックとして，LAMP がある．LAMP は，Linux（OS）, Apache（Web サーバ）, MySQL（データベース）, PHP（実行エンジン）で構成される．LAMP スタックのように複数のレイヤで構成されているシステムで障害が発生した際には，ログファイルからエラーメッセージを手掛かりに障害の原因を特定する．しかし，根本の障害が別のレイヤに波及することにより，障害の原因があるレイヤ以外のレイヤにおいてもエラーメッセージが生成されることがある．このような場合，どのレイヤに根本的な障害の原因があるのかを特定することが困難になる．例えば，MySQL で障害が発生した場合に，データベースにアクセスできないことにより上位レイヤである PHP レイヤでもエラーメッセージが生成されるようなケースがある．その結果，PHP レイヤで発生した多数のエラーメッセージにより， MySQL レイヤにある根本的な原因の存在を特定しづらくなる．実際に，LAMP スタックにおける複数のレイヤにまたがる障害の解決が困難で時間のかかる作業であることが報告されている [1].

このように，複数のコンポーネントで構成されるシステムにおいて，ログメッセージから障害原因を特定する作業のコストは大きい．そこで本研究では，サーバサイドアプリケーションで生じた障害の根本的な原因を示すログメッセージを特定するために，プログラムコード中の障害原因を特定する技術であるフォルトローカライゼーション技術をログに適用する手法を提案する．

本研究で用いるフォルトローカライゼーション技術について 2 章で説明し，3 章で障害原因を特定する手法を示す．4 章では考察を述べ，5 章で関連研究を紹介する．最後に 6 章で本稿をまとめる．

2. フォルトローカライゼーション

ソフトウェアで生じる障害の原因特定が困難になったことにより，障害原因特定のプロセスの効率化の需要が高まっている．このような背景からフォルトローカライゼーション技術は盛んに研究されるようになった．

1 慶應義塾大学
Keio University
2 南山大学
Nanzan University

ⓒ 2021 Information Processing Society of Japan

表 1　Ochiai による疑惑値算出と原因箇所特定の例

#	プログラムコード	test1 (x= 1)	test2 (x=-1)	test3 (x= 0)	Fail(s)	Pass(s)	疑惑値	順位
1	`def closeToZero(x):`							
2	`  a = 0`	✓	✓	✓	1	2	0.6	3
3	`  if x < 0:`	✓	✓	✓	1	2	0.6	3
4	`  a = x + 1`		✓		0	1	0	6
5	`  elif x > 0:`	✓		✓	1	1	0.7	2
6	`  a = x + 1`	✓			1	0	1.0	1
	`  //↑本来は a = x - 1`							
7	`  return a`	✓	✓	✓	1	2	0.6	3
	テスト結果	Fail	Pass	Pass				

表 2　FLSF が有効な例

#	プログラムコード	test1 (mylist=0∼100)	test2 (mylist=-50∼100)	test1 実行回数	test2 実行回数
1	`def sumOfAbs(mylist):`	✓	✓	1	1
2	`  ans = 0`	✓	✓	1	1
3	`  for x in mylist:`	✓	✓	101	101
4	`  if x > 0:`	✓	✓	101	101
5	`  ans = ans + x`	✓	✓	100	50
6	`  else:`				
7	`  ans = ans - 0`	✓	✓	1	51
	`  //↑本来は ans = ans - x`				
8	`  return ans`	✓	✓	1	1
	テスト結果	Pass	Fail		

2.1 Spectrum-Based Fault Localization (SBFL)

SBFL は，テストケースによる実行経路情報を利用するフォルトローカライゼーション技術である [2]．これは，失敗したテストケースで実行されたプログラム文が不具合の原因箇所である可能性が高いというアイデアに基づく．テストケース実行によるテスト結果と実行経路情報から，各ステートメントに対して疑惑値を算出し，その疑惑値により不具合の原因箇所を推定する．疑惑値の算出方法にはさまざまなものが提案されているが，一例として原因箇所特定性能が優れているとされる Ochiai の計算式では次のように定義されている [3]．

$$Suspiciousness(s) = \frac{Fail(s)}{\sqrt{TotalFail \times (Fail(s) + Pass(s))}}$$

ここで，$TotalFail$ は失敗テストケースの総数を表し，あるステートメント s を実行するテストケースのうち，$Fail(s)$ が失敗したテストケースの数を，$Pass(s)$ が成功したテストケースの数を表す．表 1 に，簡単な Java プログラムとテストケースに対して，Ochiai を利用した場合の疑惑値算出と，それに基づく原因箇所特定の例を示す．この例では，2 列目に示した Python プログラムコードの 6 行目に不具合箇所が存在する．3 ∼ 5 列目は，それぞれ個別のテストケースとそれに対する実行経路を表しており，各行が実行されたかどうかを ✓ で示している．6, 7 列目はこれらの実行経路情報を集計した，それぞれのステートメントが実行された失敗したテストケースの数・成功したテストケースの数を示す．これらのテスト実行の情報から Ochiai の計算式により算出した疑惑値を 8 列目に示している．この例では，6 行目の疑惑値が最も高く，不具合原因の可能性が高いことを示している．

2.2 Fault Localization based on Statement Frequency (FLSF)

上記の SBFL では適切に特定ができないケースがあり，このようなケースを補完するために Shu らは FLSF [4] という SBFL を拡張した手法を提案した．表 2 に，既存の SBFL では適切に特定ができないケースのプログラムの例を示す．このプログラムは mylist 内の整数の絶対値の合計を計算するプログラムで 7 行目に誤りがある．これに対して mylist に含まれる整数のリストが異なる 2 つのテストを用いて SBFL を実施する．実行経路に注目すると，3, 4 列目に示すようにステートメントごとの差異がないため，疑惑値は全ステートメントで同じ値になる．それに対し，5, 6 列目ではステートメントごとに差異が見られる．これは，各ステートメントが 1 回のテスト実行で実行され

表 3　ログファイルに SBFL を適用した例

#	ログメッセージ	ログファイル 1	ログファイル 2	ログファイル 3	Fail(s)	Pass(s)	疑惑値	順位
1	Startup.	✓	✓	✓	1	2	0.6	2
2	[info] XXX	✓	✓		1	1	0.7	1
3	Warning YYY	✓	✓		1	1	0.7	1
4	Note: ZZZ			✓	0	1	0	4
	.....							
	ログファイルの種類	障害発生時	正常稼働時	正常稼働時				

た回数を記している．FLSF は，この実行回数の分だけ重み付けを行って疑惑値を算出する．元々の SBFL は各ステートメントがそれぞれのテストケースで実行されたか否かを 1 か 0 の 2 値で評価するが，FLSF は実行回数を双曲線正接関数を用いて 0 ∼ 1 の範囲で正規化する．これにより実行回数の分だけ重み付けされるため，正規化された値を用いて疑惑値計算を行うと，失敗した時に実行される割合が高い 7 行目の疑惑値が最も高くなる．

3. サーバサイドアプリケーションの障害原因を特定する方法

3.1 概要

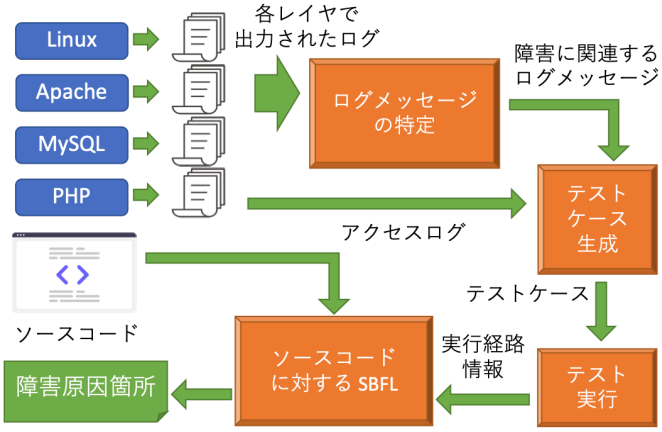

図 1　障害原因特定の概要

本研究では，LAMP スタックを対象に，ログファイルとプログラムを利用して，運用中に発生した障害の原因を特定する手法を提案する．これを実現するための核として，プログラムコード中の障害原因箇所を特定するフォルトローカライゼーションをログファイルに適用するアイデアを用いる．

障害原因特定の概要を図 1 に示す．手順としては，まず障害原因に関連するログメッセージを特定し，次に特定したログメッセージを利用することでアプリケーションレイヤに存在するプログラムコード中の障害の原因箇所を特定する．ログメッセージの特定は，障害発生時と正常稼働時に発生したログファイルに対して SBFL を適用することで実現する．プログラムコード中の障害の原因箇所を特定するためには，まず障害発生時の状況を再現する必要がある．特定した障害に関連するログメッセージの情報を利用することで，障害発生時のアクセスログから障害時の状況を再現するための情報を抽出し，これをテストケースとする．このテストケースを実行し，その際の実行経路を用いて SBFL を実施することで，プログラムコード中の障害原因箇所を特定する．

障害原因に関連するログメッセージを特定する手法の詳細を 3.2 節で，アプリケーションレイヤに存在するプログラムコード中の障害の原因箇所を特定する手法の詳細を 3.3 節で説明する．

3.2 ログメッセージの特定

アプリケーションの運用中に発生した障害に対して，各レイヤで発生したログファイルから，障害の原因に関連するログメッセージを特定する．本ステップでは以下の 2 つのアプローチで関連ログメッセージを特定していく．

- アプローチ A -SBFL 手法のログファイル適用-
- アプローチ B -レイヤ間メッセージの関連の識別-

アプローチ A で大量のログメッセージからの障害に関わるログメッセージのローカライズを行い，アプローチ A で差別化できなかったログメッセージをアプローチ B で得られた情報を用いてより詳細にローカライズする．

3.2.1 アプローチ A -SBFL 手法のログファイル適用-

アプローチ A では，本研究のアイデアの核となる SBFL 技術のログファイル適用を行う．SBFL は成功したテストケースと失敗テストケースの実行経路情報を用いることで，プログラムコード中に含まれる障害原因箇所を特定する技術である．これをログファイルに適用することで，大量に存在するログメッセージの中から，障害に関連するログメッセージを特定する．SBFL をログファイルに適用するために，以下のような対応づけを行った．

- 対象：プログラムコード → ログファイル
- 用いる情報：失敗・成功テストケースにおいて任意のステートメントが実行されたか否か
  → 障害時・正常稼働時のログファイルにおける任意のログメッセージの存在有無
- 出力：コード内の不具合原因箇所
  → 障害の原因に関連するログメッセージ

この対応づけに従って，ログファイルに SBFL を適用した例を表 3 に示す．2 列目のログメッセージが，それぞれの

表 4　FLSF を導入して出現回数を考慮したログファイル SBFL 適用例

#	ログメッセージ	ログファイル 1	ログファイル 2	ログファイル 3	疑惑値	順位
1	Startup.	1	1	1	0.6	3
2	[info] XXX	1	1	0	0.7	2
3	Warning YYY	100	1	0	0.72	1
4	Note: ZZZ	0	0	1	0	4
	.....					
	ログファイルの種類	障害発生時	正常稼働時	正常稼働時		

```
[19-May-2021 09:33:30 UTC] Startup.
[19-May-2021 09:35:11 UTC] [info] XXX
[19-May-2021 09:40:34 UTC] Warning YYY
[19-May-2021 09:40:55 UTC] Warning YYY
...
[19-May-2021 09:44:02 UTC] Warning YYY
[19-May-2021 09:44:30 UTC] Warning YYY
...
```

図 2　ログファイル 1（障害発生時）

```
[17-May-2021 09:31:30 UTC] Startup.
[17-May-2021 09:32:19 UTC] [info] XXX
...
[17-May-2021 10:02:24 UTC] Warning YYY
...
```

図 3　ログファイル 2（正常稼働時）

ログファイルに含まれているかどうかを，3 ∼ 5 列目に ✓ で示している．それらの情報から，元の SBFL 同様に疑惑値算出を行い，順位づけをする．その結果，2 行目と 3 行目のログメッセージが障害発生時に発生しやすいことが分かる．

SBFL を適用するにあたり，失敗・成功テストケースの実行経路情報を障害時・正常稼働時のログファイルに任意のログメッセージの存在有無として対応づけた．そのため，障害発生時のログファイル 1 が図 2 の内容である場合，それぞれのログメッセージの発生回数が異なるが，表 3 の 3 列目ように発生回数によらず，ただ存在の有無のみを評価している．これはつまり同一ログファイルにおけるログメッセージの出現回数をまったく考慮していないことになる．

出現回数を考慮すべきかどうかを議論するために，図 2 と図 3 を例として考える．障害発生時に多数発生しているログメッセージと 1 回しか発生していないログメッセージがある場合，発生回数の多いログメッセージの方が注目度が高くなる．例えば，障害発生時のログファイル 1 が図 2，正常稼働時のログファイル 2 が図 3 の内容である場合，正常稼働時のログファイル 2 では各ログメッセージが 1 回ずつしか発生していないが，障害発生時のログファイル 1 では，Warning YYY が多数発生している．このような場合，Warning YYY がより障害原因に関連性が高いログメッセージであると考えられる．つまり，ログファイルにおけるログメッセージの出現回数は考慮すべき情報であることが分かる．

このように，ログメッセージは発生頻度に着目する必要があるが，既存の SBFL にそのまま適用しようとすると出現回数が反映されない．そこで，ログメッセージの出現回数を考慮した SBFL を実施するために，FLSF を導入した．FLSF は，1 回のテスト実行で実行される回数の分だけ重み付けを行って疑惑値を算出する．そこで，この疑惑値計算方法をログファイルに対する SBFL に適用することで，同一のログファイル内で複数回発生するログメッセージに出現回数の分の重み付けを実現する．

以上をふまえて最終的にログファイルに SBFL を適用する手法を説明する．手順は以下のようになる．

(1) それぞれのログファイル内に含まれるログメッセージの出現回数を抽出
(2) ログメッセージの出現回数とログファイルの種類 (正常稼働時のものか障害発生時のものか) から，各ログメッセージの疑惑値を算出
(3) 疑惑値順に各ログメッセージを順位づけ

FLSF の計算方法を導入した SBFL 適用例が，表 4 である．表 3 では ”[info] XXX” と ”Warning YYY” が差別化できていなかったが，FLSF の計算方法を適用することで，出現回数に応じた重み付けを実現でき，その結果 ”[info] XXX” より ”Warning YYY” の方がより怪しいという結果を得ることができた．

3.2.2　アプローチ B -レイヤ間メッセージの関連の識別-

アプローチ B では，複数のレイヤ間に現れるログメッセージ同士で関連があるものを見つける．これは共通する文字列を含むログメッセージが複数のレイヤにまたがることがあるためである．例えば，Web アプリケーションにおいて，MySQL で障害が発生したときに，MySQL サーバの出力メッセージがそのままアプリケーションレイヤのログメッセージとして以下のように出力されるケースがある．

MySQL ログ：
Aborted connection 15 to db: (Got a packet bigger than ’max_allowed_packet’ bytes) ....

アプリケーションログ：
Database Error: Got a packet bigger than ’max_allowed_packet’ bytes for query ....

アプリケーションログが MySQL サーバが出力したメッセージをそのまま用いているため，この二つのログメッセージは"Got a packet bigger than 'max_allowed_packet' bytes" の部分が共通する．このような共通文字列部分を抽出することで，ログメッセージ同士の関連を見つける．

共通部分の抽出にはドットマトリクス法 (ドットプロット法)[5] を用いる．ドットマトリクス法は，2 つの DNA 配列間の類似性などを視覚化する手法で，以下の手順で実現する．

(1) 比較する 2 つの配列をそれぞれ 行と列のラベルとして設定
(2) 行と列のラベルが一致するセルをプロット
(3) マトリクスの主対角線（右下）方向にドットが連なっているセルを判別

表 5 は文字列 cXYZba と ababXYZcc の類似性をドットマトリクス法を用いて視覚化したものである．これにより主対角線（右下）方向にドットが連なっているセルのラベルである ba と XYZ が共通することが視覚的に分かる．このドットマトリクス法を拡張し，以下の手順で最長の共通部分を特定する．

(1) 比較する 2 つの配列をそれぞれ 行と列のラベルとして設定
(2) 各セルを以下の条件で算出
- 行と列のラベルが一致しないセルは 0
- 行と列のラベルが一致するセルは左上のセルの値に 1 加算した値

(3) マトリクス内の最も大きい値から左上方向にトレースバック

表 6 は表 5 と同じ文字列 cXYZba と ababXYZcc を用いて，各セルを上記の条件で算出したマトリクスである．このマトリクスの最大値である 3 のセルから，1 になるまで左上にトレースバックすると，この時のセルのラベルは XYZ となり，XYZ が最長の共通部分であることが分かる．

共通するメッセージとして抽出する文字列は，一定以上の長さの文字列長が共通する組み合わせに制限する必要がある．これは，単に共通する文字列だけを対象にすると，単語レベルで共通部分が抽出されてしまい，肝心の関連するログメッセージの識別が実現できなくなるためである．共通部分の文字列長を一定以上に定義することで，センテンス単位で共通するログメッセージの組み合わせのみを判別する．

3.2.3 障害原因に関連するログメッセージの特定

ログファイルに SBFL を実施した結果や，共通するメッセージ部を検出することで得られたログメッセージ間の結びつきを用いて，障害原因に関連するログメッセージを特定する．障害原因を示すログメッセージが障害発生時にのみしか発生せず，障害に関係のないログメッセージが障害発生時にあまり発生しないような場合，SBFL の結果に

表 5 ドットマトリクス法

	a	b	a	b	X	Y	Z	c	c
c								●	●
X					●				
Y						●			
Z							●		
b		●		●					
a	●		●						

表 6 最長共通部を求めるためのマトリクス

	a	b	a	b	X	Y	Z	c	c
c	0	0	0	0	0	0	0	1	1
X	0	0	0	0	1	0	0	0	0
Y	0	0	0	0	0	2	0	0	0
Z	0	0	0	0	0	0	3	0	0
b	0	1	0	1	0	0	0	0	0
a	1	0	2	0	0	0	0	0	0

よって障害原因を示すログメッセージが綺麗にローカライズされる．このような場合には，SBFL の結果のみを参考に障害原因の特定が実現できるが，そうではないケースでは他の情報を踏まえて包括的に分析する．例えば，特定したいログメッセージと同じタイミングで発生するログメッセージが多い場合には，多数の不必要なログメッセージが似たような疑惑値の値を示すため，特定したいログメッセージが埋れてしまう．このような SBFL の結果だけではすぐに特定することが困難な場合に，アプローチ B で特定した複数のログメッセージに共通するメッセージ部の情報を用いて，埋れていたログメッセージをローカライズする．また，複数のレイヤで障害原因に関連するログメッセージがローカライズされる場合には，レイヤの上下関係の情報を用いて，より下層のレイヤのログメッセージを優先的に調査していく．これは，レイヤの上下関係があるシステムにおいて複数のレイヤでエラーが生じた場合には，より下層のレイヤが根本的な障害の原因である可能性が高いためである [1].

3.3 障害原因箇所の特定方法

3.2 節で障害原因に関連するログメッセージを特定したのち，障害発生時の状況を再現するテストケースを生成する．障害の原因がアプリケーションレイヤにある場合には，アプリケーションレイヤのプログラムコードに対して SBFL を実施することで，プログラムコード中に存在する障害原因箇所を特定できる．フォルトローカライゼーションにおいて，テストケースは非常に重要なファクターであり，フォルトローカライゼーションの精度はテストケースに大きく依存する．加えて，テストケースは通常開発段階で作られるため，そのテストケースによって発見できなかった障害が運用中に初めて顕在化することがある．そのため実際の運用中に発生した障害事例を再現することがで

きるテストケースの生成は非常に有益である．

プログラムコード中の障害原因箇所の特定は以下の手順で実現する．

(1) テスト対象とするアクセスログメッセージを選択し，それを再現するテストケースを生成
(2) 障害発生時の状況を再現した環境でテストを実行
(3) テスト実行で得られた実行経路を用いて，PHP プログラムに対して SBFL を実施

3.3.1 テストケース生成

本研究は Web アプリケーションを対象にしている．Web アプリケーションは疎結合なアーキテクチャであり，独立したサーバサイドとクライアントサイドを HTTP を介して通信することでサービスを実現している．つまり，HTTP リクエストの情報が記録されているアクセスログをトレースすることで，サーバサイドの状況を再現することができる．

まず，3.2.3 節で特定したログメッセージからタイムスタンプを取得する．取得した時刻情報から，障害を引き起こしたと考えられるイベントの開始時間と終了時間を想定する．特定したログメッセージが生成された時点で障害は発生しているので，その時刻がイベント終了時刻とする．イベント開始時刻は終了時刻から一定時間遡った時刻に設定するが，非常に長いスパンのイベントによって障害が引き起こされるケースもあるため，注意する必要がある．こうして定義されたイベント開始時間と終了時間の間に発生したアクセスログメッセージを選択する．選択されたアクセスログから，リクエストメソッド，パス，パラメータを抽出し，状況を再現するための障害発生時のテストケースとする．また，アプリケーションレイヤに障害がある場合には，SBFL を実施するために正常稼働時の状況を再現するテストケースも同様に作成する．

3.3.2 テスト実行

テストを実行するために，作成したテストケースを用いて障害発生時や正常稼働時の状況を再現するためのリクエストを送信する．リクエストの送信先としては，プロダクション環境ではなくステージング環境を用いる．ステージング環境は可能な限り障害発生時のプロダクション環境と揃える必要があり，データベースなどはリクエストの度にスナップショットをとっておくなどの操作で対応する．しかし，特にハードウェアに近い部分が原因の障害は再現することが不可能なため，そのような障害は対象外とする．また，テスト実行中にサーバサイドで実行したプログラムコードの実行経路を，3.3.3 節の SBFL 実行のために記録しておく．

3.3.3 コード内での障害原因箇所の特定

アプリケーションレイヤに障害がある場合には，生成したテストケースの実行中に得られた実行経路を用いて，SBFL を実施する．SBFL は，成功するテストケースと失敗するテストケースを用いるが，本アプローチでは以下のように定義する．

- 成功したテストケースの実行経路 →
  正常稼働時の状況を再現したテストケースの実行経路
- 失敗したテストケースの実行経路 →
  障害発生時の状況を再現したテストケースの実行経路

前述のように SBFL を実施するためにテストケースは必須だが，常にテストケースが用意されているわけではないため，SBFL を実施するための最大の障壁がテストケースの存在である．しかし，本アプローチでは実際の運用中の操作をテストケースとして利用するため，SBFL の実施が可能となり，PHP 中の障害発生の原因となるプログラムコード箇所を特定できる．

4. 考察

提案手法では，障害発生時に生成されたログメッセージを過去のログメッセージの発生状況と合わせて分析することで，ログメッセージの障害に対する関連度を算出し，障害に関連するログメッセージを特定していく．そのため，十分な量の過去のログ情報を入力としてとることができれば，精度の高いログメッセージのローカライズが実現できると考えられる．逆に，過去のログファイルが十分に用意できない場合や，環境が大きく変化したことにより過去のログファイルが参考にならない場合には効果を十分に発揮しない．つまり，本提案手法の有効性は用意するログファイルに対する依存度が高いことに留意する必要がある．また，障害発生時に原因特定の役に立たないような余計なログメッセージが多数生じる場合，SBFL のアプローチだけでは十分にログをローカライズできないことが予想される．そのため，より効率的なログメッセージの特定のために，他のアプローチも合わせて検討していくべきと考える．

本提案におけるプログラムコード内の障害原因の特定は非常に効果的であると考えられる．3.3 節で述べたように，SBFL の実施における最大の障壁はテストケースの存在である．SBFL の精度はテストケースに大きく依存し，高いカバレッジを持つテストケースが十分に用意できれば，高い性能を発揮する．しかし，実際の開発環境ではテストケースは十分に用意されないことが多い．そのため，実際の障害事例を再現したテストケースは非常に有益であり，そのテストケースを使用して実施した SBFL の結果は，障害原因の特定・解決のために有益な情報を提供することが期待される．

5. 関連研究

大規模なソフトウェアは膨大なログを生成するため，障害原因に関連するログメッセージを見つけるには多大な労力を要する．この面倒な作業の一部を自動化することでデバッグのサポートをする技術がログ解析である．

ログファイルを解析するためには，まずログメッセージを構文解析する必要があるが，この構文解析を行うツールであるログパーサは多数存在する．Zhu らは，13 のログパーサについて，それぞれの精度や堅牢性，処理時間について報告している [6].

Kim らは，既存のログ解析について連続ログ解析と非連続ログ解析の 2 種類に分類している [7]．前者はシステム運用時のシステムログやクラウドログなどの連続するログメッセージの分析に焦点を当てており，後者はテストシナリオの実行中に生成されたログメッセージなどの非連続ログの分析に焦点を当てている．連続ログ解析は，過去に発生したログメッセージを用いることで有用な情報を見つけ出し，将来の故障を予測する [8][9][10]．非連続ログ解析は，テストシナリオ実行時に発生したログメッセージを解析することで失敗したテストシナリオの障害原因を特定する [7][11]．Amar らはテスト実行が成功した時のログメッセージと失敗した時のログメッセージの差分を取るアプローチをとり [11]，Kim らはテストシナリオ毎の出現頻度を利用するアプローチを提案している [7].

今日，社会で実際に運用されているシステムのほとんどは複数のコンポーネントで構成されている．構成するコンポーネントの量が多ければ多いほど，発生するログメッセージの量は増え，障害に対処するためのログ解析は困難になる．また，複数のコンポーネントで構成されるシステムにおいては，各コンポーネントで生成されるログを照らし合わせて解析する必要があるが，ログファイルは各コンポーネントごとに独立して生成されるため，さまざまな問題が生じる．それらの問題に対処するために，尾花らはソフトウェアとインフラストラクチャをシームレスに障害原因調査するフレームワークを提案した [12].

6. おわりに

本稿では，サーバサイドアプリケーションで生じた障害の根本的な原因を示すログメッセージを特定するために，プログラムコード中の障害原因を特定する技術であるフォルトローカライゼーション技術をログに適用する手法を提案した．今後，本提案の妥当性を検証するために，実際に報告された障害事例を再現し，その障害原因を本提案で特定できるかを評価する予定である．

謝辞

本研究の成果の一部は，科研費基盤研究 (C)20K11758，2021 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] Sayagh, M., Kerzazi, N. and Adams, B.: On cross-stack configuration errors, *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, pp. 255–265 (2017).

[2] Wong, W., Gao, R. et al.: A Survey on Software Fault Localization, *IEEE Trans. on Software Engineering*, Vol. 42, No. 8, pp. 707–740 (2016).

[3] Abreu, R., Zoeteweij, P. et al.: An Evaluation of Similarity Coefficients for Software Fault Localization, *2006 12th Pacific Rim Int'l Symposium on Dependable Computing (PRDC'06)*, pp. 39–46 (2006).

[4] Shu, T., Ye, T., Ding, Z. and Xia, J.: Fault localization based on statement frequency, *Information Sciences*, Vol. 360, pp. 43–56 (2016).

[5] Gibbs, A. J. and McIntyre, G. A.: The diagram, a method for comparing sequences: Its use with amino acid and nucleotide sequences, *European journal of biochemistry*, Vol. 16, No. 1, pp. 1–11 (1970).

[6] Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z. and Lyu, M. R.: Tools and benchmarks for automated log parsing, *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, pp. 121–130 (2019).

[7] Kim, J., Savchenko, V., Shin, K., Sorokin, K., Jeon, H., Pankratenko, G., Markov, S. and Kim, C.-J.: Automatic abnormal log detection by analyzing log history for providing debugging insight, *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, pp. 71–80 (2020).

[8] Du, M., Li, F., Zheng, G. and Srikumar, V.: Deeplog: Anomaly detection and diagnosis from system logs through deep learning, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1285–1298 (2017).

[9] Gurumdimma, N., Jhumka, A., Liakata, M., Chuah, E. and Browne, J.: Towards detecting patterns in failure logs of large-scale distributed systems, *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IEEE, pp. 1052–1061 (2015).

[10] Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G. and Mueen, A.: Logmine: Fast pattern recognition for log analytics, *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 1573–1582 (2016).

[11] Amar, A. and Rigby, P. C.: Mining historical test logs to predict bugs and localize faults in the test logs, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, pp. 140–151 (2019).

[12] 尾花将輝，花川典子：システムログとアプリケーションログを用いた障害起因特定のためのフレームワーク提案，コンピュータソフトウェア， Vol. 38, No. 3, pp. 3_58–3_74 (2021).