Vol.2020-SE-206 No.9
2020/11/27

故障再現のための実行メソッド履歴取得における負荷削減の検討

徐　振宇[†]　野元　励[†]　田村　高廣[†]

† NTT ソフトウェアイノベーションセンタ
E-mail: †{shinu.jo.uv,tsutomu.nomoto.cb,takahiro.tamura.zd}@hco.ntt.co.jp

あらまし　企業で利用されているアプリケーションソフトウェアは正常に運用継続することが求められており，不具合対応は必要不可欠である．故障解析は不具合対応の中で重要かつ難易度の高い作業となっており，現状行われている故障解析作業では，対象ソフトウェアに対する深い理解と多くの作業工数が必要となっている．本研究は，アプリケーションの運用中に発生するソフトウェア不具合の故障解析にフォーカスし，故障解析を容易にするための方式の実現を目標としている．具体的には、アプリケーションの運用中に常にメソッドの実行履歴を取得し、故障発生時に実行されていたメソッド呼び出しシーケンスに関して、ステートメントレベルで網羅するようにテストケースを生成及び実行し、故障を自動的に再現させることで、故障解析に必要な情報を容易に取得可能とする方式を目指している。本稿では，アプリケーションの運用中にメソッド実行履歴を取得する際の負荷削減に取り組み、取得対象メソッドをコールグラフの深さで選択することで、負荷を最大 1.8 倍程度まで抑えられることを確認した．

キーワード　アプリケーション運用，効果的な計測，コールグラフ，故障再現

Study of load reduction in the execution method history acquisition for the failure reproduction

Xu ZHENYU[†], Nomoto TSUTOMU[†], and Tamura TAKAHIRO[†]

† NTT Software Innovation Center
E-mail: †{shinu.jo.uv,tsutomu.nomoto.cb,takahiro.tamura.zd}@hco.ntt.co.jp

Abstract　Troubleshooting is an essential part of enterprise-level software maintenance and operation. Failure analysis plays an important role for troubleshooting. But at the same time failure analysis is a highly difficult task, which requires a deep understanding of the target software. This research focuses on failure analysis of software defects that occur during application operation, which aims to realize a method for facilitating failure analysis. More specifically, the execution history of the method is always acquired during the operation of the application, and the test case is generated and executed so as to cover the method call sequence that was executed at the time of failure at the statement level, and the failure is automatically performed. In this paper, we worked on reducing the load when acquiring the method execution history during application operation, and confirmed that the load can be suppressed up to about 1.8 times by selecting the method to be acquired by the depth of the call graph.

Key words　Application Maintenance, Efficient Instrumentation, Call Graph, Failure Reproduction

1. はじめに

1.1 背　景

企業で利用されているアプリケーションソフトウェアは正常に運用継続することが求められており，不具合対応は必要不可欠である．本研究では、企業の社内システムとして利用されているような、常に正常な動作が求められるアプリケーションの不具合対応を対象としている。不具合発生時のユーザからの問い合わせ対応は、図 1 に示すような流れを想定し、ユーザにサービス提供している環境 (本番環境) と異なる環境として、不具合対応で再現試験等の解析に使用する環境 (検証環境) を用意できることを前提としている。

一次解析では、自然言語による発生状況、画面キャプチャ、ログ等のユーザからの申告情報を基にマニュアルや既知故障等のナレッジを確認し、その範囲で解決や回答を行う。一次解析で解決出来ない新規の故障等の場合は二次解析が実施される。二次解析では、一次解析同様

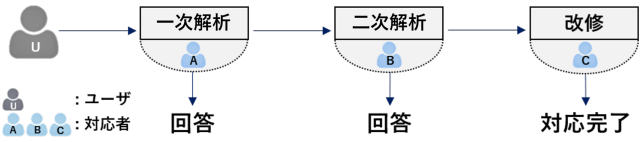

図 1　ユーザからの問い合わせ対応の流れ

にユーザからの申告情報を基に、ソースコード解析、再現試験等により故障解析を実施する。故障解析により原因が特定されると、故障の改修が実施される。

二次解析で行われる故障解析では、再現試験により故障を再現することで故障解析に必要となる情報を得られるが、申告情報から発生条件が特定出来ず、故障再現が難航すること場合がある。その場合、作業者は仮説と検証を繰り返しながら、ソースコード解析や再現試験を行うこととなり、対象ソフトウェアに対する深い理解と多くの工数が必要となる。

2. 故障解析の既存技術及び問題点

この節では，故障解析の現状を述べた後，既存技術とその問題点を述べる．

2.1 故障解析の現状

図 2 に，故障の流れを説明する．本研究における故障とは、ソースコードにある欠陥からスタートして，欠陥があるコードの実行により影響が広まり，最終的にユーザが気付く障害 [1] となる、一連の流れを指す。JSTQB [12] によると，欠陥は「コンポーネントまたはシステムに要求された機能が実現できない原因となる、コンポーネントまたはシステムに含まれる不備」と定義されている．

本稿における故障解析とは、ソースコード上の欠陥個所と，処理開始から障害に至るまでのフロー (以降では故障パス) を明らかにすることである．故障パスの具体例としては，故障発生時に実行されていた一連のステートメントやメソッドの呼び出しシーケンスである．

二次解析の作業者は、対象アプリケーションのソースコードや、ユーザからの申告情報として受領したログや自然言語による発生状況、画面キャプチャ等から、検証環境での再現試験により詳細な情報を取得した後に、ソースコード等により欠陥個所の推定を試みる。しかし、ユーザから申告情報として受領したログには、実行メソッドの履歴等の故障パスが記録されていないことが多く、申告情報から故障を再現することが難しい場合がある。その場合、作業者は故障に繋がる複数の仮説を立て，複数パターンの検証環境の準備や再現試験の試行錯誤を行うことが必要となり、対象ソフトウェアに対する深い理解と多くの工数が必要となる。

このため，現状の作業者による仮説と検証を繰り返す方法ではなく，故障解析に要する工数や故障解析のハードルを下げる新たなアプローチが必要である．

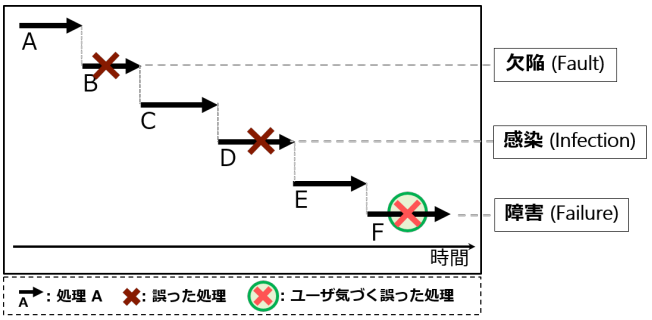

図 2　故障の流れ

2.2 対象とする故障

本研究では、2.1 節の記載に該当する故障及び故障解析を想定しており、呼び出しシーケンスがソースコードのステートメントレベルで正常時と異なるものを対象としている。資源のリークや設計誤りのような、正常時と呼び出しシーケンスの違いが検出できないものは対象に含めない。

2.3 既存技術及び問題点

本節では、既存の故障解析技術、故障再現技術を挙げる。また、既存技術について本番環境への適用観点を中心とした問題点を挙げる。本番環境に適用するためには、本番環境への性能負荷が小さいこと、現実的な時間で再現が可能であることが必要な条件となる。

欠陥箇所を推定する故障解析技術としては，アプリケーション開発時に用いられる故障原因推定技術 [2] がある．故障原因推定技術の中で故障パスの取得に最も近いスペクトルベースの特徴と問題点を説明する．

スペクトルベース (*Spectrum Based*) [3]:

> ステートメントや制御分岐ブロック粒度で情報取得の対象を指定し，成功した実行と失敗した実行から得られる情報の差分 (ステートメントレベルの実行履歴など) を用いて，欠陥の場所を推定する技術である．このような手法 [4] では情報取得の対象が多く負荷が大きい問題と，現実的な時間で網羅的に膨大な数のテストケースを作成することが難しい問題があり、本番環境に適用することは難しい．

また，アプリケーション運用時に発生する故障を対象とした *TimeTravel* [7], [8] のような故障再現技術もある．

TimeTravel:

> *TimeTravel* は、ユーザからの入力等のアプリケーションへの外部入力データを用いて，故障再現を実施する手法である．この手法は、プログラム内部の処理に紐付いた解析を行っていないため，故障の再

図 3　提案する全体方式のイメージ

現性が非決定的である．このため再現できない場合が多く発生すると考えられ，本研究で対象としている故障の解析手法として利用することは難しい．

また *TimeTravel* の中で，再生可能スタックトレース [10] による故障を再現する研究もある．プログラムに例外が発生した際に，その瞬間のプログラム内部の情報を細かく出力することで，例外発生時の実行を再現する技術である。この手法は、メモリ使用量が大きく増加する問題があり、本番環境に適用することは難しい。また、情報取得のタイミングが例外発生時に限定されるため、正常時と異なる呼び出しシーケンスであるが例外が発生しないような場合は対応できない。この問題は一般的なスタックトレースでも同様である。

3. 問題解決に向けた提案と課題

3.1 提案する全体方式の紹介

本研究では、本番環境で適用できる故障解析を実現するために、以下の方式を検討している。

本番環境での情報取得に関しては、既存の故障原因推定技術 [2] で行われていたステートメントレベルからメソッドレベルに粒度を粗くして実行履歴を取得することで、情報取得量及び負荷を削減する。故障発生後は、本番環境から取得したメソッドレベルの実行履歴について、ステートメントレベルで網羅するようにテストケースを生成し、検証環境でテストを実施することで故障を再現させる。故障再現により、故障発生時のステートメントレベルの故障パスを取得する。故障解析の作業者は、故障発生時のステートメントレベルの故障パスから、ソースコード解析の実施や既存の故障原因推定技術 [2] を適用することで、欠陥箇所を特定する。

故障を再現するまでの全体方式のイメージを図 3 に示す．全体方式は、以下 3 つのステップに分けられる．

Step1: トレース対象の絞り込み

ソースコードや例えば Java のバイトコードのような中間コード等を用いてアプリケーション全体の構造を解析する．その解析結果を用いて，情報を取得する対象 (以降はトレース対象) のメソッドを絞り込んで，トレース対象のリストを生成する．

Step2: 実行時の情報取得

Step1 で生成したトレース対象のリストを用いて、トレース対象の実行履歴 (以降は実行痕跡) が取得できるように本番環境で実行するアプリケーションを変更し，アプリケーション実行時に情報を取得する．トレース対象に全てのメソッドを選択していた場合は、実行痕跡は故障パスと同一になる。トレース対象のメソッドが絞り込まれるた場合は、実行痕跡は歯抜けのある故障パスとなる。

Step3: 故障の再現

Step2 で収集した実行痕跡とコールグラフ等のメソッドの呼び出し関係の情報から、メソッドレベルの故障パスを復元する．実行痕跡は対象メソッドが絞り込まれており、歯抜けのある故障パスとなるため、故障パスは必ずしも 1 通りに決まらず、想定される故障パスは複数パターンとなり得る。実行痕跡から想定される故障パスについて、以降では故障パス候補と呼ぶこととする。メソッドレベルの故障パス候補に関して，ステートメントレベルで網羅したテストケースを作成する．作成したテストケースを検証環境で実行し，故障を再現させる．

3.2 課題及び解決策の検討

3.2.1 課　　題

本提案方式の実現に向けては，Step1,2 における実行痕跡の取得における適切なトレース対象の選定手法を確立することが必要である．

運用中のアプリケーションから実行痕跡を取得すると性能負荷が必ず生じる．トレース対象の数が少ないと負荷が小さくなるが、故障パス候補数が増え、作成及び実行するテストーケースも多くなるため、故障再現にかかる時間が長くなる。トレース対象の数が多いと負荷が大きくなるが，故障パス候補数が少なくなり，作成及び実行するテストケースが少なくなるため、故障再現にかかる時間が短くなる。既存の故障原因推定技術 [2] では，ステートメントや制御分岐ブロック単位等の細かい粒度で故障パスを取得することになるため，負荷が莫大なものとなっている．検証環境では負荷が許容される場合もあるが，本番環境ではサービス利用に影響するほどの負荷は許容されないため，本番環境に適用可能な程度に負荷を削減する手法を確立する必要がある．

3.2.2 実行痕跡の取得手法

本研究の具体的な手法としては，業務システムから WEB アプリケーションに至る様々な領域で広く用いられている Java を対象として検討を進める．

アプリケーション運用中に情報を取得する一般的なアプローチとして、ログ出力を設定することが挙げられる．*log4j* [9] は Jakarta プロジェクトで開発が進められている

Java プログラム用のログ API である．設定ファイルにおいてログ出力のレベルを設定すれば，それより高いレベルのログはすべて出力される．一般的なアプリケーションのログ出力の使い方としては，「DEBUG」レベルに設定することで，詳細な実行時の情報を出力させることができる。しかし，ログの出力箇所と内容は開発時のソースコードで決められているため，元々アプリケーションが持っているログ出力だけでは、必ずしも故障解析に必要な情報が取得できる保証がない．

そこで本研究では、実行痕跡を取得するためのログ出力を対象アプリケーションの処理として追加することで、情報取得を実現する。まず最初に、アプリケーションのメソッド呼び出し関係を解析するため，コールグラフを作成する．Java のコールグラフを作成するため，本研究では対象アプリケーションを静的解析するツール java call graph [11] を利用する．次に、作成したコールグラフからトレース対象とするメソッドを絞り込み、トレース対象のメソッドにのみ実行痕跡を出力するためのログ出力処理を追加する。

3.3 負荷削減に向けたアプローチ

本節では、最初にコールグラフを用いて，メソッドレベルの一意な故障パスへの復元を担保できるように，トレース対象を削減する方法紹介する．本研究で利用するコールグラフは，メソッドをノードとして，プログラムが通り得るメソッド呼び出しの全経路をグラフ化したものである．次に，更なる負荷削減効果を得るために提案する手法を説明する．

3.3.1 一意なメソッドレベルの故障パスを取得する方法

最初に、対象のアプリケーションの全てのメソッドの実行履歴を記録するシンプルな方法を紹介する．続いて，それをベースにトレース対象を削減する方法を説明する．

全メソッド inout:

全てのメソッドに対し，メソッド開始時 (以降は in) とメソッド終了時 (以降は out) にスタンプ情報 (メソッド名と時刻) を出力する方法である．この方法では，一意なメソッドレベルの故障パスを取得することができるが，全メソッドで情報取得が 2 回必要となり，負荷が大きい．

全メソッド in:

全てのメソッドの in と，コールグラフから判別できる一部の特殊な out だけ取得することで，一意なメソッドレベルの故障パスを取得することができる．大部分の特殊な out がないメソッドでは，情報取得が 1 回となるため，全メソッド inout よりも負荷が削減できる．

特殊な out については、図 4 で示す。図中の番号はメソッド名を示しており、メソッド 2 の out が上記の特殊な out である．

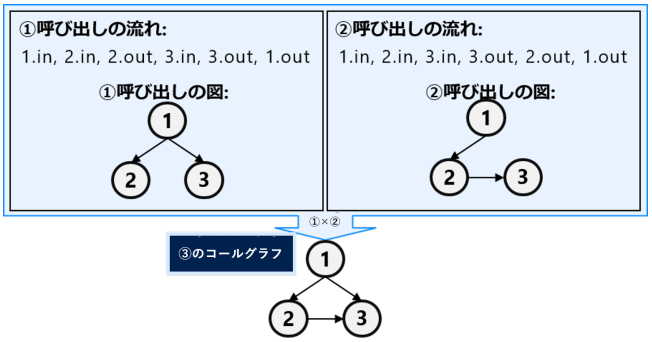

図 4　特殊な out の説明例

図 4 でこの結論について説明する．各メソッドの in のみ記録した時，”1.in”,”2.in”,”3.in” が記録されたとする．このとき，あり得る呼び出しは図 4 に示した①と②二通りである．そして，二通りの呼び出しの可能性を合成し，③で示したコールグラフとなることがあり得る．つまり，③の構成の下で in のみを記録すると，判別ができなくなる．ここで，呼び出し列 (1 → 2 → 3 となる) にある 2 番目のメソッドの out を記録すれば，区別ができるようになる．この例からメソッド数を増やし，呼び出し列を伸ばしても，区別できないコールグラフは全部①と②の組み合わせに集約される．

更なるトレース対象を削減する方法として，Mustafa らの研究がある [5]．トレース対象の数を減らすため，Mustafa らはドミネーターツリーを利用した．ドミネーターツリーは，ノードが基本ブロック (ステートメントレベルに相当する粒度) であり，親ノードは子ノードを支配 (ルートから子ノードに辿る経路はならなず親ノードを経由すること) するグラフである．ドミネーターツリーの特徴は，ドミネーターツリー上の葉ノードをトレースすれば，メソッドの呼び出しシーケンスが分かることである．

しかし，この手法でトレース対象は基本ブロックであり，メソッドではない．トレース対象をメソッドに変えて適用すると，下記の問題が生じる．葉ノードをトレース対象にすると，葉ノードより手前のメソッドで終了する処理の場合は何の実行痕跡も残すことができない．この問題はドミネーターツリーに限ず、少なくとも全メソッドの in 情報を取得しなければ，メソッドの呼出しシーケンスの欠落部分が生じるため，一意なメソッドレベルの故障パスを保障できなくなる．

以上より，更なるトレース対象の削減は，呼び出しシーケンスから一意なメソッドレベルの故障パスを取得することができなくなる．更なるトレース対象の削減を実現ためには、実行痕跡から複数の故障パス候補を出し、故障パス候補を網羅するようにテストケース作成及びテスト実施を行う必要がある．故障パス候補の数が増えるとテストケース作成及び実施の時間が長くなるため，故障

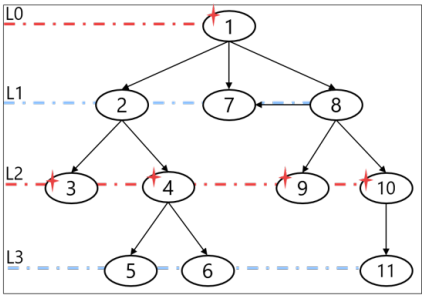

図 5　サンプルコードのコールグラフ

パス候補を絞る方法が必要である．

3.3.2　提案手法：選択メソッド

提案する選択メソッドのアルゴリズムを説明する．コールグラフ表現でルートメソッド (0 段) から深さが隔段毎となるように，メソッドの in をトレース対象として選択する．例えば，自作したサンプルコードのコールグラフである図 5 では，トレース対象が L0 段目の”1.in”, と L2 段目の”3.in”,”4.in”,”9.in”,”10.in” となる．

続いて，上記の選択方法とした理由を記載する．

a. 実行時間の増加の偏りが無く，負荷を均一化することが期待できる．これはコールグラフ表現での深さをベースにトレースを実施するため，どの経路で実行してもトレース対象の数が均一となることが見込めるからである．

b. コールグラフの経路によらず，故障パス候補数を均一化することが期待できる．故障パスの不確かな範囲は，隔段で同様に発生するため，故障パスの候補数も均一となることが見込めるからである．

コールグラフでは，グラフの途中で呼び出しシーケンスが終了する場合が表現できないことや，ソースコード上の詳細な判定条件によっては通らない経路もグラフに表現されることがあるが，コールグラフはトレース対象の選択のために利用しており，Step3 で故障再現を行う際は，ソースコードから正確な故障パス候補が確認できる．

4. 評 価 実 験

実行痕跡の取得手法について評価を行った．

評価実験の対象は以下の条件から選定した．アプリケーションのソースコード修正が必要であるため，ソースコードが入手可能であること．負荷を計測するため，大規模 (コールグラフの深さ，メソッドの数) なソフトウェアであること．これらの条件から，NTT 社内で開発、利用されている大規模アプリケーションを対象として，その中の 1 機能を用いて評価を行った．

4.1　評価項目の説明

従来の故障原因推定技術手法と 3.3 節で挙げたアプローチを用いて，比較評価を実施した．以下の各手法について対象アプリケーションのある 1 機能の呼び出し開始から終了までの処理時間を計測し，比較した．

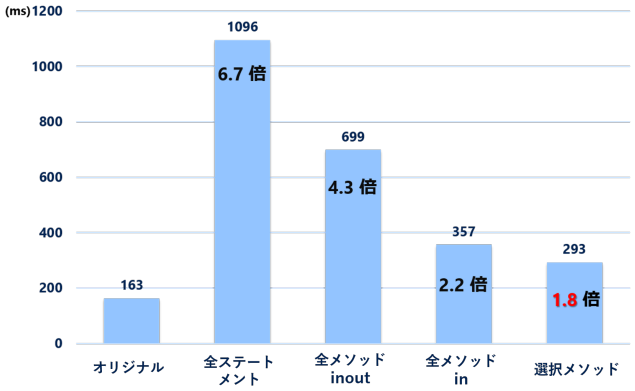

図 6　処理時間の比較

オリジナル: プログラムに変更を加えていない処理である．この処理時間をベースとして，他の手法との比較を行う．

全ステートメント: 従来の故障原因推定技術であるスペクトルベースの手法で用いられているトレース対象を選択する方法であり，各ステートメントに毎に情報を出力する．

全メソッド inout: 3.3.1 の「全メソッド inout」に該当する方法である．各メソッドの実行が開始する時と終了する時に情報を出力する．

全メソッド in: 3.3.1 の「全メソッド in」に該当する方法である．今回の評価対象では特殊な out が存在しなかったため，各メソッドの実行が開始する時に情報を出力する．

選択メソッド in: 3.3.2 の「選択メソッド」に該当する方法である．コールグラフ表現でルートメソッドから深さが隔段毎となるようにトレース対象を選択し，選択されたメソッドの実行が開始する時に情報を出力する．

4.2　評 価 結 果

評価結果を図 6 に示す．一意なメソッドレベルの故障パスを特定できる方法としては、オリジナルの実行時間を 1 にした場合，**全ステートメント**は 6.7 倍，**全メソッド inout** は 4.3 倍，**全メソッド in** は 2.2 倍である．故障パス候補が複数となる提案手法である、**選択メソット in** では**オリジナル**の実行時間の 1.8 倍である．

5. 考　　察

メソッドレベルの故障パスを一意に特定するアプローチとしては，**全メソッド in** が限界であることが分かる．更なる負荷削減のアプローチとして，本研究では**選択メソッド in** を提案した．提案手法は、**全メソッド in** よりも負荷を削減することができたが、故障パスを一意に特

定することができないため、故障再現に必要となる時間が増加し、負荷と故障再現に必要な時間はトレードオフ関係となっていることが分かる。

また、故障パス候補数はコールグラフの深さに比例して増加される特徴があるため、コールグラフの深さに伴って故障再現に必要となる時間も増加していくことが見込まれる．今回の実験対象で試算した場合，故障パス候補数は平均 39 本であり，再現時間は故障パスを一意に特定できるアプローチより最悪 39 倍程度かかることが考えられる。

また，**選択メソッド in** を改善して効率よく負荷を削減するアプローチとして、例えば，一定期間の運用実績に応じ，メソッドの呼び出し回数を考慮してトレース対象を選定することも考えられる．

6. まとめ及び今後の課題

6.1 ま と め

本研究では，アプリケーションの運用中に発生するソフトウェア不具合の故障解析にフォーカスし，本番環境に適用可能な故障解析を容易にするための方式を検討した。特にアプリケーションの運用中にメソッド実行履歴を取得する際の負荷削減について検討及び評価した。実験により、メソッドレベルの故障パスを一意に特定できる手法では負荷が 2.2 倍が限界であることを確認し、提案手法では取得対象メソッドをコールグラフの深さで隔段ごとに選択することで、故障の再現時間は増加が見込まれるが、取得負荷については 1.8 倍まで削減できることを確認した。

6.2 今後の課題

提案する全体方式では，故障パスをメソッドレベルで取得するため，欠陥箇所の推定に必要なステートメントレベルの情報に呼び出しシーケンスが含まれていない．そのため，取得したメソッドレベルの故障パスを制御フローグラフに適用し，ステートメントレベルの構造を網羅するようにテストケースを作成し，故障を再現する手法を確立する必要がある．

故障再現のアプローチとして，動的記号実行 [6] を考えている．アプリケーションの全ての構造をステートメントレベルで完全に網羅するテストケースを作成しようとすると数が莫大となり，作成及び実行にかかる時間が現実的なものとならないが、メソッドレベルの故障パスあるいは故障パス候補により、故障再現のために作成するテストケースを絞ることで，現実的な時間での実現を考えている．しかし，このアプローチでテストケースを作成する場合，下記の問題を検討する必要がある．ステートメントの構造を網羅するようにテストケースを追加する場合，ループを除外する必要がある．このため、ループにより生じる故障について対策を検討する必要がある．

文 献

[1] Andreas Zeller, "デバッグの論理と実践 なぜプログラムはうまく動かないのか", O'REILLY, 2012
[2] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, "A Survey on Software Fault Localization," IEEE TSE, 2016, pp. 707-740
[3] J. S. Collofello and L. Cousins, "Towards automatic software fault localization through decision to decision path analysis," in Proc. Nat. Comput. Conf., 1987, pp. 539-544
[4] V.Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in Proc. Eur. Conf. Object Oriented Program., Glasgow, U.K., 2005, pp. 528-550
[5] Mustafa M. Tikir, Jeffrey K. Hollingsworth, "Efficient Instrumentation for Code Coverage Testing," ISSTA, 2002, pp. 86-96
[6] 高松 宏樹, 佐藤 晴彦, 小山 聡, 栗原 正仁,"動的記号実行によるメソッドの複雑度を考慮したテストケース自動生成,"IPJS-SIGSE, 2014-SE-185(27), 1-7, 2014
[7] Yu Zhao, Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Wei Zheng, William G.J. Halfond, "ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports," ICSE, 2019
[8] Thomas Bach, Ralf Pannemans, Johannes Haeussler, Artur Andrzejak, "Dynamic Unit Test Extraction Via Time Travel Debugging for Test Cost Reduction," ICSE-Companion, 2019
[9] log4j HOMEPAGE, https://logging.apache.org/log4j/2.x/
[10] 宗像 聡, 梅川 竜一, 上原 忠弘, "再生可能スタックトレースによる Java Web アプリケーションの例外発生過程の調査", J-STAGE, 2019, pp. 101-118
[11] java-callgraph tool github page, https://github.com/gousiosg/java-callgraph
[12] "ソフトウェアテスト標準用語集（日本語版）", http://jstqb.jp/dl/JSTQB-glossary.V2.3.J02.pdf