特別研究報告

題目

継続的インテグレーションにおける実行トレースの取得及び変数に関する変化情報の出力を行うツールの試作

指導教員
井上 克郎 教授

報告者
藤原　勇真

令和 4 年 2 月 8 日

大阪大学 基礎工学部 情報科学科

令和 3 年度 特別研究報告

継続的インテグレーションにおける実行トレースの取得及び変数に関する変化情報の出力を行うツールの試作

藤原　勇真

内容梗概

現在，ソフトウェア開発において，VCS(バージョン管理システム) が広く利用されている．VCS を用いた開発では，CI(継続的インテグレーション) ツールの活用が進んでおり，継続的なコードの統合や，自動的なテストの実行が行われている．VCS 及び CI ツールを用いることで，ソースコードの変更箇所や，テストスイートをクリアするか否かは目に見えるが，プログラムの挙動がどの程度変化したかは分からない．効率的なデバッグにおいて重要となるのが，プログラムの挙動の観測であるように，開発者はプログラムの挙動の変化に注意すべきである．

そこで本研究では，CI 上で実行トレースの取得を行い，プログラムの挙動の変化として，開発者へ変数に関する変化情報の出力を行うツールの試作を行なった．このツールでは，値が変わった変数や追加・削除された変数の数が多い場合，プログラムの挙動の変化が大きいと判断し，プログラムの挙動に注意するよう開発者へ警告を行う．このツールにより開発者は，ソースコード上の変化だけでなく，プログラムの挙動の変化に注意することができる．また，試作ツールを用いて，調査を行なった．

調査では，GitHub 上にある 5 つのプロジェクトに対し，ソースコードの変更量と変数に関する情報の変化量の関係について調査を行った．調査の結果，5 つのプロジェクトのうち，1 つがやや正の相関関係あり，2 つがかなり正の相関関係あり，2 つがほとんど相関関係なしという結果が得られた．このことから，ソースコードの変更量と変数に関する情報の変化量の相関は必ずしも強いわけではないということが分かる．ソースコード上の変化だけでなく，プログラムの挙動の変化として変数に関する変化情報を開発者に見せることは，有用であると言える．

主な用語

継続的インテグレーション
実行トレース
バージョン管理システム

目 次

1　まえがき

現在，ソフトウェア開発において，バージョン管理システム（Version Control System:以下，VCS）が広く利用されている．VCS を用いた開発では，開発の本流であるブランチから新たに派生ブランチを作成し，派生ブランチで各々が作業を行い，成果物が完成した後に本流のブランチに統合 (マージ) する. これによって，複数の開発者が同時に開発を進めることが可能となり，効率的なソフトウェア開発が実現できる.

また，VCS を用いた開発では，自動的・定期的にテスト，ビルド，検証が行える，継続的インテグレーション（Continuous Integration:以下，CI）のツールの活用が進んでいる．これによって，派生ブランチでの変更箇所を定期的に本流のブランチにマージしたり，編集する度に自動的にテストを実行することが可能となり，より効率的なソフトウェア開発が実現出来る．

ソースコードを変更した場合，当然プログラムの挙動が変化する．VCS において代表的な Git の場合，編集したファイルをコミット・プッシュした際，ソースコードの変更箇所はを差分の形で表示することができるため，ソースコード上の変化を見ることは容易である．また，CI ツールを活用することで，プッシュ後のプロジェクトがテストスイートをクリアするか否かも確認することが出来る．しかし，プログラムの挙動がどのように変化したかは確認することが出来ない．開発者の予期せぬ挙動の変化があったにも関わらず，ソースコード上の変化及びテストケースの結果から気づけなかった場合には，今後のプログラム開発に大きな影響を及ぼす可能性がある．そこで，CI ツールを活用することで，ソースコード上の変化だけでなくプログラムの挙動の変化を開発者へ見せることはできないか，またプログラムの挙動の変化が大きい際，開発者へ通知することが出来ないか，と考えた．

本研究では，プログラムの挙動の変化を示すことを目的とし，CI 上で実行トレースの取得及び開発者へ変数に関する変化情報の出力を行うツールの試作を行なった．今回は VCS として Git を利用し，Git リポジトリマネージャとして GitLab を利用する．このツールでは，GitLab へプッシュ前後の実行トレースをそれぞれ取得し，実行トレースから変数の情報を抽出し，変数に関する変化情報を出力する．実行トレースの取得には，限られた保存領域を利用し実行トレースの取得を行う NOD4J[14] を利用した．また，変数に関する情報の変化量が大きい場合，プログラムの挙動が大きく変化していると判断し，プログラムの挙動に注意するよう開発者へ警告を行う．この試作したツールによって，プログラムの挙動がどの程度変化したのか，開発者はプログラムの挙動の変化に注意することが出来る．

以下，2 章では研究背景，3 章では提案手法について説明する．4 章では，ツールの実装について説明する．5 章では，試作ツールを用いて行う調査について説明する．6 章では試作したツールの問題点について，7 章では本研究のまとめについて説明する．

図 1: コミット前後のコードの変更箇所

2 背景

2.1 VCS を用いたソフトウェア開発

VCS とは，ファイルの変更履歴の保存，管理を行うソフトウェアである．ファイルの内容を，いつ誰がどのように編集したのか，時系列で記録に残すため，過去のあるバージョンを参照したり，過去のバージョンに戻したり，過去のバージョンとの差分を取ることが可能である．VCS には，CVS（Concurrent Versions System）や，SVN（Subversion）などの集中型バージョン管理システムと，Mercurial や Git などの分散型バージョン管理システムがある．本研究では，分散型である Git を利用する．Git では，開発の本流であるブランチから派生ブランチを作成し，派生ブランチで各々が作業を行った後，各々の成果物を本流のブランチに統合（マージ）する，といった開発手法が取られている．このようにして，複数の開発者による効率的なソフトウェア開発が可能となる．

Git では，変更したファイルをコミット・マージした際，ソースコードの編集箇所を開発者が確認できる．本研究で使用する GitLab の場合，図 1 のように，全体での変更されたファイルの数，追加された行数，削除された行数及び，各ファイルの変更箇所が確認できる．赤でハイライトされたコードが削除され，緑でハイライトされたコードが追加され，拡張子 “.EC” に対する条件式が追加されたことが分かる．このように，開発者はソースコード上の変化を確認することが出来る．

2.2 CI ツール

CI とは，ビルド，テスト，検証を繰り返し行うことで，問題の早期発見及び開発の効率化を図る手法である．CI を実際に手動で行うと非常に手間がかかるため，一般的には，自

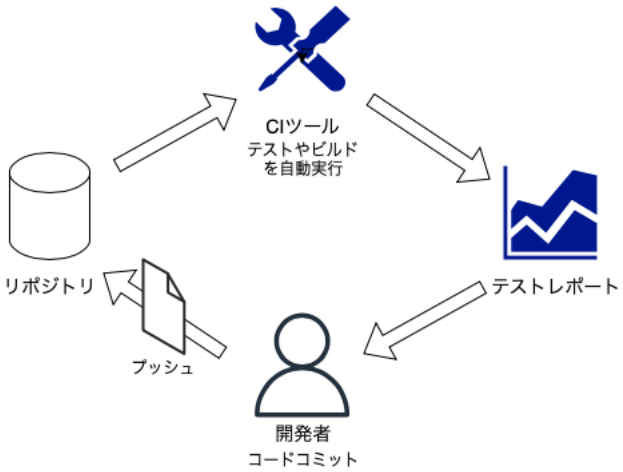

図 2: CI ツールを用いたソフトウェア開発の流れ

動的にこれらのプロセスを実行する CI ツールが用いられている．CI ツールを用いたソフトウェア開発の流れは図 2 のようになる．まず開発者が変更したファイルをリポジトリにコミット・プッシュする．次に CI ツールが稼働し，変更後のソースコードに対して，ビルド，テストを自動的に実行する．その後，開発者は CI ツールから得られたテストの結果や解析ツールの結果等のレポートを参考に，再び開発を行う．

近年，この CI ツールの導入は進んでおり，多くのプロジェクトで利用されている．Hilton らの調査結果によると，約 34,000 件の OSS プロジェクトのうち，約 4 割のプロジェクトで CI を導入していた [8]．この CI ツールには，テストやビルドだけでなく，メトリクスを測る解析ツールなど，様々なツールを組み込むことが可能である．また，CI ツールを用いることで，プロジェクトの欠陥を早期に発見することができる．Vasilescu らの研究により，CI を導入している OSS プロジェクトは，そうでないプロジェクトより多くの欠陥を検出できていることが分かっている [15]．

CI ツールを用いることで，プロジェクト内の多くの欠陥を発見することができる．しかし，プロジェクト内の全ての欠陥を発見できるわけではない．南の研究によると，CI を実施している 246 件の OSS プロジェクトのうち，自動テストにより 28 件のプロジェクトで 332 個の欠陥が発見されたが，そのうちの 7 件のプロジェクトで 17 個の欠陥が見逃されていた [16]．CI 上のテストで発見されなかった欠陥については，ソースコード上の変化及びプログラムの挙動の変化から発見しなければならない．

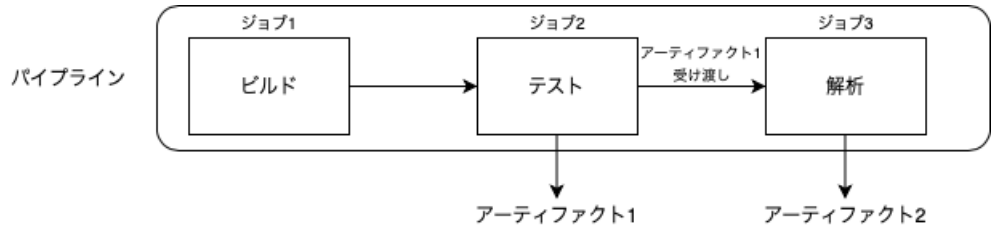

図 3: パイプラインの例

2.3 GitLab CI

本研究で使用する GitLab の CI ツールについて説明する．GitLab の CI ツールが実行する一連の流れをパイプラインと呼び，パイプライン内のそれぞれの実行単位をジョブと呼ぶ [5]．ジョブでは，実行するスクリプトを定義する．また，ジョブによって生成されたファイルやディレクトリのことをアーティファクトと呼ぶ．同一パイプライン内にあるジョブ間では，容易にこれらのアーティファクトの受け渡しが可能である．パイプラインの内容は，.gitlab-ci.yml ファイルを用いて設定する．

パイプラインの例を図 3 に示す．ビルド，テスト，解析を行うそれぞれの実行単位がジョブであり，全体がパイプラインである．今回の例の場合，テスト，解析を行う各ジョブからアーティファクトが生成されている．生成されたアーティファクトは GitLab 上に保存される．また例のように，テストを行うジョブから得られたアーティファクトを，解析を行うジョブへと受け渡すことが可能である．

2.4 プログラムの挙動

Omniscient Debugging といった手法が提案されているように，効率的なデバッグにおいて重要となるのが，プログラムの挙動の観測である [13]．プログラムの挙動には，メソッド呼び出し，メソッド実行，変数アクセスやフィールドアクセス，それらの制御フロー等が挙げられる．多くの開発者は，変数の値を出力する printf デバッグを有用な手段だと考え，利用している [7]．また，プログラムの挙動を観測するために，処理の状況を示すメッセージや変数の値等のデータをプログラムの外部に出力するロギング処理が広く用いられている [11]．プログラムの実行を任意の箇所で一時停止するブレークポイントデバッグも頻繁に利用されており [12]，変数の値やメソッドの呼び出し状態など，プログラムの挙動が観察されている．

開発者は VCS と CI を用いることで，ソースコード上の変化やテストスイートにクリアするか否かが分かる．しかし，プログラムの挙動がどのように変化したかは確認することができない．コードの編集箇所が少ない場合でも，誤ってコードを編集していた場合，プログラ

ムの挙動が大きく変化し，バグが発生する可能性がある．また，編集時にバグが発生せずとも，プログラムが予期せぬ挙動をしていた場合には，後々のデバッグが困難になる可能性がある．

開発者は，コードの変更量だけでなく，プログラムの挙動の変化，特に変数に関する情報の変化について注意する必要がある．

2.5 Omniscient Debugging

Omniscient Debugging とは，プログラムの実行中のメモリ状態を時系列で完全に記録することで，網羅的に情報を収集してデバッグする手法の 1 つである [9]．この手法は任意の時点でのプログラムの内部状態を計算機上で再現し，命令の実行順序や変数の値を観測することが可能である. 本研究では，Omniscient Debugging の実装の 1 つである NOD4J[14] を使用する．

2.6 NOD4J

NOD4J とは，Java プログラムを対象とした，テストケースに対する実行トレースの取得及び可視化を行うデバッグツールである．実行トレースとは，網羅的にプログラムによって実行された命令とそれによる値の変化である．NOD4J は，実行トレース取得コンポーネント，実行トレースとソースコードのマッチングコンポーネント，可視化コンポーネントの 3 つで構成されている．NOD4J では，プリミティブ型及び String 型の変数の実行トレースを取得できる．本研究では 3 つのコンポーネントのうち，実行トレース取得コンポーネント及び実行トレースとソースコードのマッチングコンポーネントを利用する．

3　提案手法

本研究では，GitLab へプッシュ前後のプログラムの挙動の変化を示すことを目的とし，CI 上で実行トレースの取得及び変数に関する変化情報の出力を行うツールの試作を行う．今回は，プログラムの挙動の変化として変数に関する情報に着目する．2.4 節でも述べたように，開発者はプログラムの挙動として変数に着目することが多く，開発者にとって変数に関する情報が最も有用ではないかと考えたからである．本章では，まず 3.1 節で試作したツールを CI に組み込んだソフトウェア開発の全体の流れについて説明する．次に 3.2 節では，GitLab CI の設定について，3.3 節では実行トレースの取得方法について説明する．3.4 節では，パイプライン間のアーティファクトの受け渡しについて，3.5 節では，変数に関する変化情報の取得・出力方法について説明する．最後に 3.6 節で開発者への警告について説明する．

3.1　全体の流れ

本研究で試作したツールを CI に組み込んだ際の全体の流れは図 4 のようになる．プッシュ前に実行されたジョブから実行トレースを取得するために，開発者はまず GItLab CI 上の JOB_ID を管理するサーバを起動する (1)．次に，開発者が変更したファイルをリポジトリにコミット・プッシュする (2)．開発者が手動で行うのはこの 2 つの操作のみであり，これ以降の動作は自動的に実行される．まず GitLab CI が起動され (3)，コンテナが起動される (4)．コンテナ内でテストが実行され，同時に NOD4J を用いてプッシュ後の実行トレースの取得が行われる (5)．取得した実行トレースの情報は，ジョブのアーティファクトとして，GitLab 上に保存される．次に，プッシュ後の実行トレースの取得を行なったジョブの JOB_ID をサーバに送信し (6)，プッシュ前の実行トレースの取得を行なったジョブの JOB_ID をサーバから取得する (7)．取得した JOB_ID からプッシュ前の実行トレースの情報を取得し (8)，試作ツールを用いてプッシュ後の実行トレースとの差分を取得する (9)．最後に，変数に関する変化情報を開発者に出力し，変化量が設定した閾値を超えた場合には，プログラムの挙動に注意するよう警告メッセージを出力する (10)．

3.2　GitLab CI の設定

GitLab の CI ツールを使用するには，ジョブを実行するオープンソースソフトウェアである GitLab Runner が必要となる．GitLab Runner には複数の実行形態があるが，今回は Docker executor を利用する．Docker とは，コンテナ型の仮想環境を作成、配布、実行するためのプラットフォームである [3]．Docker executor を使用することで，Docker のコンテナを利用することが可能となる．今回は，Docker のコンテナを用いて，実行トレースの取得や JOB_ID の送受信，変数に関する変化情報の取得及び出力を行う．

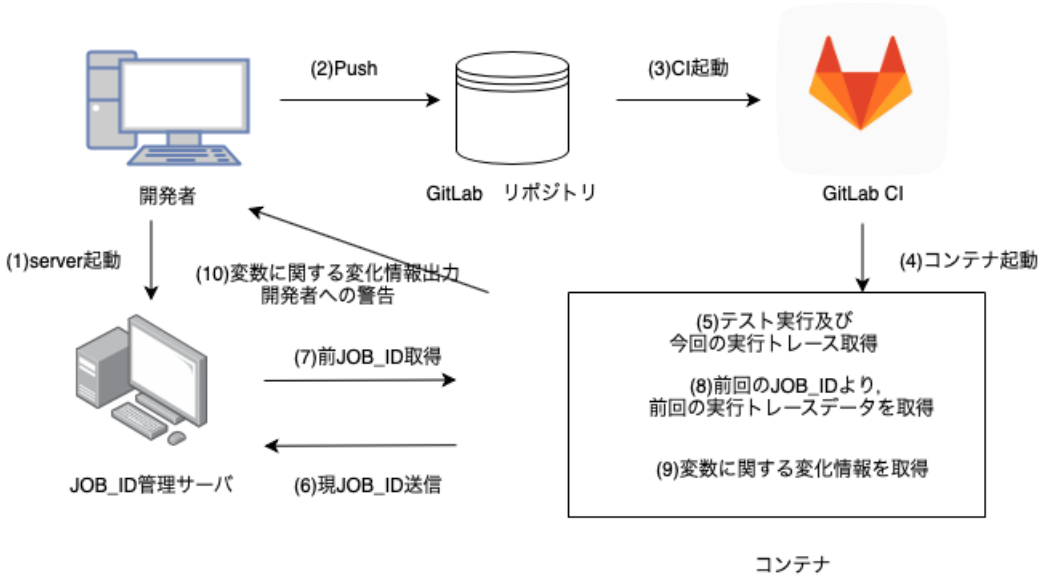

図 4: 提案手法全体の流れ

3.3　実行トレースの取得

今回は，NOD4J の実行トレース取得コンポーネントである SELogger[10] を使用し，テストケース実行時に実行トレースを取得する．CI 上で，実行トレースを取得するジョブを実行する．

3.4　パイプライン間でのアーティファクトの受け渡し

2.3 節でも述べたように，同一パイプライン内にあるジョブ間では，容易にアーティファクトの受け渡しが可能である．しかし，各パイプラインは独立しており，また各ブランチで並列にも実行される可能性があるため,「直前のパイプラインのアーティファクトを取得する」ような操作は標準では用意されていない．直前のパイプラインのアーティファクトを取得するには，ジョブに一意に割り当てられている JOB_ID 及び GitLab の API を利用する必要がある．本研究では, プッシュ後に実行されるパイプラインから，直前に実行されているパイプライン内のアーティファクトを取得する．つまり異なる 2 つのパイプライン間でアーティファクトを受け渡しするため，JOB_ID 及び GitLab の API を利用する必要がある．そのため，JOB_ID を管理するサーバを構成する必要がある．

3.5　変数に関する変化情報の取得・出力

変数に関する情報は，NOD4J の実行トレース取得コンポーネント及び実行トレースとソースコードのマッチングコンポーネントを使用することで得られる，varinfo.json ファイルか

ら取得する．varinfo.json は図 5 のような基本構成で変数情報を格納している．dataid は，各変数に割り当てられている ID（数字）である．className はクラス名，methodName はメソッド名，var は変数名である．linenum は，その変数が位置する行数であり，inst は命令の種類である．count は，同一変数の行内での何回目の出現かを表している．valueList は変数の値をリストで管理している．

本研究では，プログラムの挙動の変化として，varinfo.json から取得可能である，変数の参照・代入回数の変化，追加・削除変数，各変数のトレース長の変化の総和，値が変わった変数情報の 4 つを取得し，開発者へ出力する．変数の参照・代入回数の変化は，プロジェクト全体の参照・代入回数をプッシュ前，プッシュ後それぞれ示し，変化を出力する．参照・代入回数の変化があったファイルは，ファイル名及びそのファイルにおける参照・代入回数の変化も同じように出力する．追加・削除変数は，プロジェクト全体の個数及び追加・削除された各変数のクラス名，メソッド名，変数名を出力する．各変数のトレース長の変化の総和は，値のみ出力する．値が変わった変数情報は，プロジェクト全体での個数及び値が変わった各変数のクラス名，メソッド名，変数名を出力する．

3.6 開発者への警告

プログラムの挙動の変化が大きいとき，開発者へ警告を行う．今回は，NOD4J で値が取得可能な変数に対し，トレースが変化した変数の数の割合がある閾値 (%) を超えた際，プログラムの挙動の変化に注意するよう，ブラウザコンソールへ警告メッセージを出力する．トレースが変化した変数の数は，追加・削除変数の数及び値が変わった変数の数の総数と定義する．今回閾値は，開発者がオプションで変更可能とし，ツール試作時にはデフォルト値を設定する．

```
{
  dataid:,
  className:,
  methodName:,
  var:,
  linenum:,
  inst:,
  count:,
  valueList: [
   {data:, timestamp:, thread:},
  ]
}
```

図 5: varinfo.json の基本構成

4　試作ツールの実装

本章では，まず 4.1 節で試作ツールの仕様について説明する．4.2 節では JOB_ID の管理方法について，4.3 節では，変数に関する変化情報の取得及び出力について説明する．4.4 節では開発者への警告について，最後に 4.5 節ではツールの実行例について説明する．本ツールの作成は，TypeScript[6] を用いて行なった．

4.1　ツールの仕様

本ツールは Java プログラムを対象としたものである．本ツールは，GitLab へ変更したファイルをコミット・プッシュすることで，開発者に以下のような変数に関する変化情報を出力する．

- 変数の参照・代入回数の変化
- 追加・削除変数
- 各変数のトレース長の変化の総和
- 値が変わった変数の情報

また本ツールでは，変数に関する情報が大きく変化した際に，プログラムの挙動が大きく変化したとし，開発者へ警告を行う．

4.2　JOB_ID の管理

3.4 節でも述べたように，プッシュ前の実行トレース情報を取得するには JOB_ID が必要となるため，JOB_ID を管理するために簡易的なサーバを構成する．本研究では，Node.js のフレームワークである Express[4] を用いて実装する．サーバでは, 現 JOB_ID と前 JOB_ID の 2 つのみを管理するとし，それぞれの JOB_ID をテキストファイルに書き込み，管理を行うとする．JOB_ID の送信及び取得については，curl コマンド [2] を利用し，GET メソッド及び POST メソッドによって，JOB_ID を送受信する．

4.3　変数に関する変化情報の取得・出力

本節では，4.1 節で述べた，開発者へ出力を行う変数に関する各変化情報の取得及び出力方法について説明する．

4.3.1 変数の参照・代入回数の変化の取得・出力

変数が参照されているか，代入されているかについては，varinfo.json ファイルの命令の種類 (inst) で区別できる．命令の種類を考慮し，変数の値の数をカウントすることで，参照・代入回数を取得することができる．参照・代入回数は，ファイル毎にカウントし，ファイル毎に管理する．プッシュ前後それぞれで参照・代入回数を求め，プッシュ前後におけるプロジェクト全体の参照・代入回数を表示する形で，変化を出力する．また，参照・代入回数の変化があったファイルは，ファイル単位での変化を同様に出力する．

4.3.2 追加・削除変数の取得・出力

本ツールでは，varinfo.json ファイルのクラス名 (className)，メソッド名 (methodName)，変数名 (var) の組み合わせで変数を区別する．まず，全変数をプッシュ前後でそれぞれ求める．次にプッシュ前後で求めた全変数をそれぞれ比較し，プッシュ前のみに存在する変数，プッシュ後のみに存在する変数，プッシュ前後どちらにも存在する変数を求める．プッシュ前のみに存在する変数が，削除された変数であり，プッシュ後のみに存在する変数が，追加された変数である．追加・削除された変数の全体の個数及び各変数のクラス名，メソッド名，変数名を出力する．

4.3.3 各変数のトレース長の変化の取得・出力

トレース長とは，変数の値の遷移 (トレース) の長さである．例えば，ある変数がプログラムの実行において 4 回代入された場合，その変数のトレース長は 4 となる．本ツールでは，各変数毎に varinfo.json ファイルの valueList を連結させ，各変数の値の遷移が確認できるように管理をしている．プッシュ前後でこのリストを比較し，各変数毎にトレース長の変化を求め，トレース長の変化量を合計する．合計した値を開発者に出力する．

4.3.4 値が変わった変数情報の取得・出力

前節でも述べたように，本ツールではクラス名，メソッド名，変数名の組み合わせで各変数毎に値を管理している．NOD4J ではプリミティブ型及び String 型の変数の値のみ取得できるため，該当する変数のみ管理する．比較対象としては，4.3.2 節で取得したプッシュ前後どちらにも存在する変数のみとし，追加・削除変数は値が変わった変数情報に含めないとする．比較方法は，各変数のトレースを 1 つずつ比べ，変数のトレースが完全に一致する場合のみ，値が変化していないと判断する．変数のトレースが異なる場合や，トレース長が変化している場合は，値が変化したと判断する．出力の仕方として，まずプロジェクト全体で

値が変わった変数の数を出力し，その後値が変わった各変数のクラス名，メソッド名，変数名を出力する．

4.4 開発者への警告

本研究では，プッシュ後のプログラムにおける NOD4J で値を得られる全変数の数に対し，トレースが変化した変数の数の割合が閾値 X%を超えた際，開発者へ警告を行う．この閾値 X は，変数に関する変化情報の取得・出力を行うツール実行時に引数として設定可能であり，デフォルト値は X=10 と設定している．3.6 節でも述べたように，トレースが変化した変数の数は，追加・削除変数の数及び値が変わった変数の数の総数である．開発者への警告は，ブラウザコンソールへメッセージを出力する形で行う．

4.5 ツールの実行例

今回は，int 型の引数を 3 つ与え，その最大値を返すプログラムを例として扱う．このプログラムに変更を加えたプログラムがソースコード 1 のようになる．まず，getMax メソッド内に，int add = 0;　というコードを追加し，String 型変数 ‘moji’ の値を “moji1”から “moji2”へと変更する．続いてテストケースにも変更を加えた．変更を加えたテストケースがソースコード 2 のようになる．このテストケースでは，assertEquals メソッドを用いて，getMax メソッドの返り値が正しい値かどうかを確認している．今回は，6 回目に呼び出している assertEquals メソッドの引数である，getMax メソッドの第一引数を 10 から 20 へと変更する．以上の変更を加え，試作ツールを使用した場合，図 6 のような出力が得られる．

まず，int 型の ‘add’ という変数が増え，代入文が 1 つ増加している．このことから，変数 ‘add’ が追加されたとし，追加変数の情報が出力されている．また，今回テストケースとして，getMax メソッドを 6 回呼び出しているため，代入回数が合計 6 回増えていることになり，変数の代入回数が 89 回から 95 回へ増えている．続いて，String 型の変数 ‘moji’ の値が “moji1 ”から “moji2 ”に変化しているため，変数 ‘moji’ のトレースが変化したとし，変数 ‘moji’ の情報が出力されている．また，テストケースにおいて，getMax メソッドを呼び出す際の第一実引数，つまり getMax メソッドの仮引数である ‘num1’ の値が 10 から 20 へと変化しているため，’num1’ のトレースが変化したとし，情報が出力されている．最後に，トレースが変化した変数の数の割合が，今回警告の閾値としてデフォルトで設定されている全体の変数の数の 10%を超えているため，プログラムの挙動の変化に注意するよう，警告メッセージを出力している．

```
package sample;

public class Main {
    public static void main(String args[]) {
    }

    private static int[] intarray = new int[100];
    public static int getMax(int num1, int num2, int num3) {
        int max = 0;
        int add = 0; //add
        String moji = "moji2";//change moji1=>moji2
        if (num1 < num2) {
            if (num2 < num3) {
                max = num3;
            } else {
                max = num2;
            }
        } else {
            if (num1 < num3) {
                max = num3;
            } else {
                max = num1;
            }
        }
        for (int i = 0; i < 100; i++) {
            intarray[i] = max;
        }
        return max;
    }
}
```

ソースコード 1: 最大値を返すプログラム

ソースコード 2: 最大値を返すプログラムに対するテストケース

```
package testsample;
import static org.junit.Assert.*;
import org.junit.Test;
import sample.Main;

public class getMaxTest {
    @Test
    public void getMaxTest1() {
        assertEquals(30, Main.getMax(30, 10, 20));
        assertEquals(30, Main.getMax(30, 20, 10));
        assertEquals(30, Main.getMax(20, 10, 30));
        assertEquals(30, Main.getMax(20, 30, 10));
        assertEquals(30, Main.getMax(10, 20, 30));
        //assertEquals(30, Main.getMax(10, 30, 20));
        assertEquals(30, Main.getMax(20, 30, 20));
    }
}
```

```
<<<変数の参照回数 : 323 => 323 >>>
<<<変数の代入回数 : 89 => 95 >>>
sample/Main.java : 89 => 95
<<<追加変数: 1 >>>
(クラス名: sample/Main.java メソッド名: getMax 変数名: add )
<<<削除変数: 0>>>
<<<変数のトレース長の変化量 : 6 >>>
<<<値が変化した変数>>>
全体 => 2
(クラス名: sample/Main.java メソッド名: getMax 変数名: num1 )
(クラス名: sample/Main.java メソッド名: getMax 変数名: moji )
!!!!!WARNING!!!!!!
プログラムの挙動の変化に注意してください
!!!!!WARNING!!!!!!
```

図 6: ツールの実行例

表 1: 調査対象の Java プロジェクト

プロジェクト名	コミット数	テストケース数
maven-dependencies-analyser	102	5
tomcat-jakartaee-migration	171	20
victims-enforcer-legacy	148	5
EventDispatcher	328	41
gitflow-incremental-builder	914	249

5 試作ツールを用いた調査

本研究では，試作したツールを用いて，既存プロジェクトの開発履歴からソースコードの変更量と変数に関する情報の変化量の間に相関関係があるかどうかについて調査する．本調査では，プログラムの挙動の変化として変数に関する変化情報が有用なものであるかどうか確認することを目的とする．

5.1 調査対象及び手法

今回の調査では，GitHub 上にある，Junit のテストケースが用意された Java プロジェクト 5 つを調査対象とする．調査対象のプロジェクト名，コミット数，最新版のテストケースの数を表 1 に示す．

調査手法について説明する．既存プロジェクトのコミット履歴からコミットを抽出し，GitLab へコミット･プッシュを行うことで，既存プロジェクトの開発履歴を辿る．今回，コミット数が比較的少ないプロジェクトである，maven-dependencies-analyser，tomcat-jakartaee-migration，victims-enforcer-legacy に対してはコミット毎に開発履歴を辿り，コミット数が比較的多いプロジェクトである，EventDispatcher，gitflow-incremental-builder に対しては，デフォルトブランチへのマージ毎に開発履歴を辿るとする．また，プロジェクトによっては，実行毎に値が変わる変数が存在する．この変数を考慮するため，同じソースコードに対し 2 度テストを実行し，得られた 2 つの実行トレース情報から，実行毎に値が変わる変数の情報を取得する．具体的には手順 1 から手順 9 のような流れで調査を行う．

手順 1. テストケースが用意されており，テストがビルド可能なコミット・マージのみを抽出する．

手順 2. プロジェクトの状態を手順 1 で抽出した各コミット・マージの状態に戻す

手順 3. GitLab へコミット・プッシュを行う．

手順 4．コードの変更量を求める．

手順 5．テストを実行し，実行トレースの取得を行う．

手順 6．試作ツールを用いて，プッシュ前後における変数に関する変化情報の取得を行う．

手順 7．再度テストを実行及び実行毎に値が変わる変数の取得を行う．

手順 8．手順 2 から手順 7 を繰り返し行う

手順 9．変数に関する情報の変化量と，ソースコードの変更量との相関関係について調べる

手順 1 は手動でテスト実行を行い，該当するコミット・マージを抽出する．手順 2 についても手動で行う．具体的には git checkout コマンドを用いて状態を戻す．手順 3 から手順 7 は Automater[1] を用いて自動で行った．Automater とは，様々なワークフローを自動化して簡単に作業を効率化できるアプリケーションである．手順 4 におけるコードの変更量は git log コマンドを用いて求める．git log コマンドでは，各ファイルにおいて追加行数と削除行数が表示される．本調査では，追加行数と削除行数の合計値をコードの変更量として定義する．ただし，ソースフォルダ内のファイルやテスト実行に関するファイルのみを対象とし，プログラムの挙動に変化を及ぼさない，md ファイルや gitignore ファイル，LICENSE ファイルなどの変更行数は除く．ここで，ソースコードの変更量が 0 の場合，そのコミット・マージの状態で得られた結果は無視するとする．今回の調査で相関関係を調べる項目は以下の 4 つである．

- 変数の参照回数の変化量
- 変数の代入回数の変化量
- 各変数のトレース長の変化量の総和
- トレースが変化した変数の数

3.6 節でも述べたように，トレースが変化した変数の数とは，追加・削除変数の数及び値が変わった変数の数の総数である．

5.2 結果

ソースコードの変更量と変数に関する情報の変化量の相関係数は表 2 のようになった．また，ソースコードの変更量とトレースが変化した変数の数の関係を散布図で表したものが図 7 から

表 2: ソースコードの変更量と変数に関する情報の変化量の相関係数

対象プロジェクト	参照回数の変化量	代入回数の変化量	トレース長の変化量の総和	トレースが変化した変数の数	トレースが変化した変数の数 (実行毎に値が変わる変数を除く)
maven-dependencies-analyser	0.356754	0.310492	0.335237	0.279604	0.3974850
tomcat-jakartaee-migration	0.633902	0.496220	0.489786	0.529670	0.519600
victims-enforcer-legacy	0.796156	0.725351	0.705359	0.706590	0.694060
EventDispatcher	0.006568	-0.008273	-0.063504	0.089519	–0.014189
gitflow-incremental-builder	0.111493	0.104430	0.110788	0.206530	0.144268

図 11 になる．maven-dependencies-analyser ではやや正の相関関係あり，tomcat-jakartaee-migration，victims-enforcer-legacy ではかなり正の相関関係あり，EventDispatcher，gitflow-incremental-builder ではほとんど相関関係なし，という結果が得られた．また，各プロジェクトの相関関係は，実行毎に値が変わる変数を除く前に比べ，大きな変化は見られなかった．

原因の 1 つとしては，テストケース自体を編集した際，少量のコード変更でもプログラムの挙動が大きく変化する場合があるという点が考えられる．また，テストカバレッジがプロジェクトにより様々であり，ソースコード内においてテストケースに対するプログラムの挙動に大きく影響を及ぼす部分，及ぼさない部分が存在するといった可能性も，原因の 1 つとして考えられる．

以上の結果より，ソースコードの変更量と変数に関する情報の変化量は必ずしも相関関係があるとは限らず，相関の強さもプロジェクトにより様々であるということが分かる．つまり，ソースコードの変更量が大きい場合でも，プログラムの挙動の変化は小さい場合や，反対にソースコードの変更量が小さい場合でも，プログラムの挙動が大きく変化する場合がある．ソースコード上の変化以外に，プログラムの挙動の変化として，変数に関する変化情報を開発者に示すということは有用であると言える．

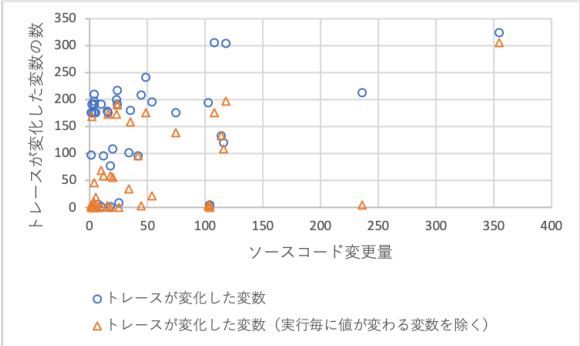

図 7: maven-dependencies-analyser
相関散布図

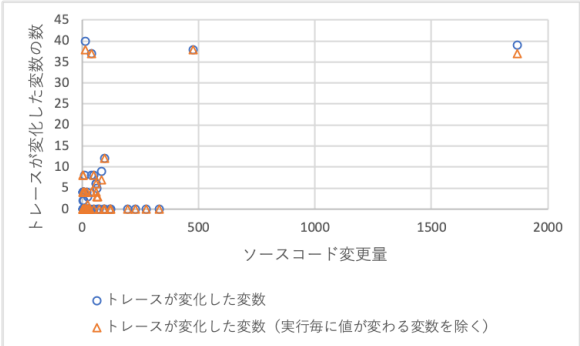

図 8: tomcat-jakartaee-migration
相関散布図

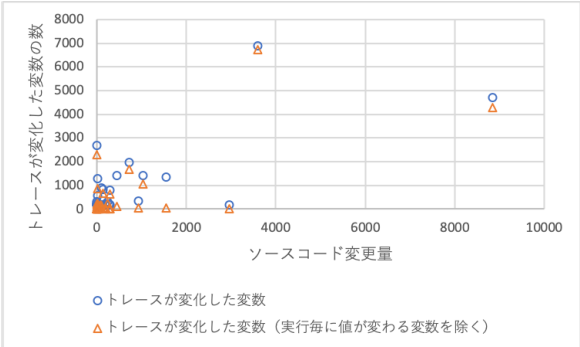

図 9: victims-enforcer-legacy
相関散布図

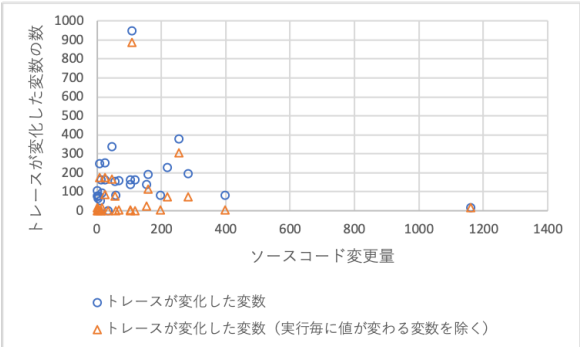

図 10: EventDispatcher
相関散布図

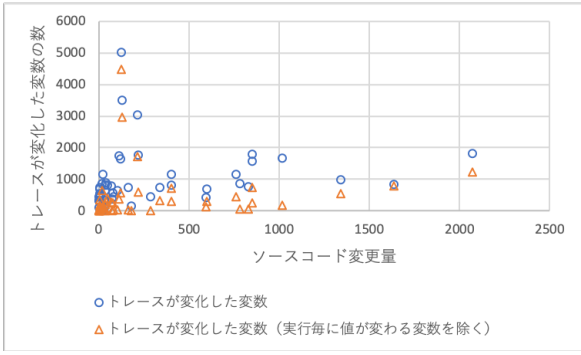

図 11: gitflow-incremental-builder
相関散布図

6　試作ツールの問題点

試作ツールの問題点として次の 3 点が挙げられる．

1 点目は，実行毎に値が変わる変数を考慮できるのか，という点についてである．試作ツールを用いた調査では，同じソースコードに対し得られた 2 つの実行トレース情報を用いて，実行毎に値が変わる変数であると思われるものを抽出した．抽出した結果，これらの変数はほとんどがプラグイン，API に属するものであった．変数名より考察すると，恐らく時間依存の変数や，毎実行毎にランダムで ID を割り当てられる変数等であると考えられる．また，実行毎に変数の数自体も変化する場合もあった．試作ツールを用いた調査と同様の操作を，CI 上に組み込めば，実行毎に値が変わる変数であると思われるものは除去できる．しかし，偶然に値が一致する可能性や，実行毎に変数の数も変化していた点を考えると，この操作で実行毎に値が変わる変数を全て除くことは難しいと考える．この点については，実行毎に値が変わる変数がどのようなものか，今後調査が必要である．

2 点目は，開発者へ警告を行う際の閾値である．今回，デフォルト値として 10%の変数の値が変わった際，警告を行うとしている．しかし実際に，変数に関する情報がどの程度変化すると，プログラム上にバグが発生する可能性が高いかなど，変数に関する情報の変化量とバグの関係性は現時点では不明である．この点についても，今後調査が必要と考える．

3 点目は，スコープの考慮である．試作ツールでは，クラス名，メソッド名，変数名で変数を区別しているため，スコープを考慮できていない．ソースコード情報と照らし合わせ，スコープの問題についても解決する必要がある．

7　まとめ

本研究では，プログラムの挙動の変化を開発者に見せることを目的とし，CI 上で実行トレースの取得及び変数に関する変化情報の出力を行うツールの試作を行なった．

試作ツールを用いた調査では，ソースコードの変更量と実行トレースの変化量間の相関関係は必ずしも強いものではなく，相関の強さもプロジェクトによって様々であるという結果が得られた．この結果より，ソースコード上の変化だけでなく，プログラムの挙動の変化として，変数に関する変化情報を開発者に見せることは有用であると言える．

しかし，実行毎に値が変化する変数が存在するという問題や，開発者へ警告を行う際の閾値をどうするかといった問題点が生じた．実行毎に値が変化する変数情報は無用なものであると考えられ，また開発者へ警告する際の閾値についても重要な点だと考えられる．

この問題点を対処するべく，今後追加調査を行い，有用性の高い変数に関する変化情報のみを開発者に見せるツールや，メソッド系列の呼び出し変化など変数以外の挙動変化情報についても出力できるようなツールを作成したい．また，情報を出力するだけでなく，サーバを用いた可視化などに発展させていきたい．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上克郎 教授には，研究活動に対して多くの貴重な御助言や御指導を賜りました. 心より深く感謝申し上げます.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には，研究室の発表機会において，御意見，御助言を賜りました. 松下誠 准教授に心より深く感謝申し上げます.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 助教には，研究活動の直接のご指導，論文の執筆に至るまで，あらゆる場面で多くのご指導を賜りました. 心より深く感謝申し上げます.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 嶋利一真 氏には，研究室での生活や，研究活動において，日々様々な御支援や御助言を賜りました. 心より深く感謝申し上げます.

最後に，その他様々な御指導，御助言等を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に，心より深く感謝申し上げます.

参考文献

[1] Automater. https://support.apple.com/ja-jp/guide/automator/welcome/mac.

[2] curl. https://curl.se.

[3] Docker. https://www.docker.com.

[4] Express. https://expressjs.com/ja/.

[5] GitLab Docs. https://docs.gitlab.com.

[6] TypeScript. https://www.typescriptlang.org.

[7] Moritz Beller, Niels Spruit, and Andy Zaidman. How developers debug. *PeerJ Preprints*, p. e2743v1, 2017.

[8] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 426–437. IEEE, 2016.

[9] Bil Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, 2003.

[10] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 253–257, 2014.

[11] Hassani Mehran. *Studying and Detecting Log-related Issues*. PhD thesis, Concordia University, 2018.

[12] Gail C Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Elipse IDE? *IEEE software*, pp. 76–83, 2006.

[13] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *ACM SIGPLAN Notices*, pp. 535–552, 2007.

[14] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, Naoto Ishida, and Katsuro Inoue. NOD4J: Near-omniscient debugging tool for Java using size-limited execution trace. *Science of Computer Programming*, p. 102630, 2021.

[15] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 805–816, 2015.

[16] 南智孝. OSS 開発における欠陥数とカバレッジの関係に基づく継続的インテグレーションの有効性検証. Master's thesis, 奈良先端科学技術大学院大学, 2017.