プログラム実行履歴を用いたリグレッションの原因検出に向けて

石部 大夢　山崎 徹郎　千葉 滋

プログラムを修正した際に今まで正しく動いていた部分が意図せず動かなくなることをリグレッションという．リグレッションを早期に検出するにはリグレッションテストが有効であるが修正するには原因を特定する必要がある．本論文ではテストが失敗した時その原因がプログラム中のどの関数にあるのかを絞り込む手法を提案する．過去のテストの実行履歴を保存し，現在の実行履歴と比較することで原因を絞り込む．本手法では二つのテストの間で振る舞いが変わった関数を失敗の原因の候補とする．この時引数などが同じ呼び出しにも関わらず結果が変わることを振る舞いが変わったとみなす．また一つの関数の振る舞いが変わるとそれを呼び出している関数の振る舞いも変わる．そのように結果が変わった関数呼び出しは候補から除外する．これにより精度は劣るものの関数ごとに多数のテストを用意しなくてもリグレッションの原因を絞り込めるようにすることを目指す．

1　はじめに

ソフトウェア開発においてしばしば機能追加やバグ修正の際に意図せず昔は正常に動いていた機能を破壊してしまうことがある．これをリグレッションと言い早期に発見及び修正することが望ましい．リグレッションを発見する既存手法としてリグレッションテストが存在する．しかし修正をする上ではどの変更がリグレッションを引き起こしたかを見つける必要があり既存手法ではそれが容易でない．

本研究ではプログラムの実行履歴を用いてリグレッションの原因となっている関数を見つける手法を提案する．この手法ではプログラムを実際に実行するコードを統合テストのテストコードのように利用し，過去のプログラムと現在のプログラムでそのコードをそれぞれ実行する．統合テストとの違いはテストコードの正解を利用者が書くのではなく過去のプログラムの振る舞いを正解とする点である．実行時の各関数呼び出しの振る舞いの差分を取ることで，リグレッションが起きているかどうかとその原因となった関数がどこであるかを突き止める．我々は提案手法を JavaScript で実装をし Bugbear と名付けた．Bugbear が正しくリグレッションの原因を提案できるかを確認するためバグベンチマークスイートを利用して実験を行った．その結果 2 つのバグケースのうち 2 つに対して正しくリグレッションの原因を提示することができた．

2　リグレッションの発見と修正

リグレッションの修正は既存の手法では容易でない．リグレッションとはプログラムを修正した際に意図せず今まで正しく動いていた部分が動かなくなることである．ソフトウェア開発においてリグレッションは早期に発見及び修正した方がいい．リグレッションの修正にはリグレッションを発見することとその原因を突き止めることが必要である．機械的なリグレッションへの対処法としてはリグレッションテストなどが知られているが，これはリグレッションの発見までのみを行い，原因の特定は支援しない．

そこで本研究ではリグレッションの修正を容易にするため，プログラム実行履歴からリグレッションの原因を特定するツールの開発を目指している．図 1 と図 2 のコード例を用いて既存のリグレッションテストでは

Concurrent Operations on Splay Trees.

Hiromu Ishibe Tetsuro Yamazaki Shigeru Chiba, 東京大学 大学院 情報理工学系研究科, Graduate School of Information Science and Technology.

```
const wordConversion = (word) => {
  const dict = {
    'かん': '勘',
    'が': 'が',
    'いい': 'いい'
  }
  return dict[word];
}

const sentenceConversion = () => {
  const words = ['かん', 'が', 'いい'];
  return words.map((word) => {
    return wordConversion(word);
  }).join("");
}
```

図 1　変更前コード

原因を見つけることが困難なことを説明する．図 1 は漢字変換を行うコードで図 2 はそれを変更した結果リグレッションを起こしているコードである．説明のため現実的なものを簡略化している．WordConversion はひらがなの単語を受け取り漢字に変換をして string 型で返す．SentenceConversion はひらがなの文を単語に区切り，単語ごとに WordConversion を呼び出して漢字変換した後，変換された単語を結合して string 型で返す．ここで，入力補完機能 InputCompletion を加えることとなったとする．この機能は変換の候補を複数要求するため WordConversion による漢字変換も図 2 のように複数の変換結果を array 型で返すよう変更する．しかし SentenceConversion は WordConversion が string 型で単一の変換結果を返す前提で作られているため WordConversion が array 型を返すよう更新した今，エラーとなる．このような変更の際に意図せず既存コードを破壊してしまうことをリグレッションという．

このリグレッションを単体テストと統合テストで検証しただけではテストが失敗することはわかるが原因を絞り込むことは難しい．まず全ての関数に単体テストが用意されている場合について考える．変更前コード用に作られた単体テストで変更後コードをテストすると変化した関数だけでなくそれを呼び出している関数も連鎖的に失敗して大量のテストが失敗して

```
const wordConversion = (word) => {
  const dict = {
    'かん': ['勘', '缶'],
    'が': ['が'],
    'いい': ['いい', '良い']
  }
  return dict[word];
}
```

図 2　変更後コード

原因の絞り込みが難しい．この例だと返す値が array 型に変わった WordConversion が失敗するだけでなく，それを呼び出している SentenceConversion，さらに SentenceConversion を呼び出している関数があればそれらも連鎖的に失敗するようになる．単体テストではなく統合テストを用いると，失敗するテストの数は減る．しかし統合テストは上位の関数だけを呼ぶので，下位のどの関数が原因であるかを絞り込むのは難しい．

リグレッションの原因を絞り込むために関数の挙動を書き出すデバッグ手法が職人芸的に知られているが，これを手動でやるには手間と知識が必要である．各関数中に引数や戻り値を出力する print 文を挿入することで実行時の各関数の挙動を書き出すことができる．これを利用して元のプログラムと変更後のプログラムでそれぞれテストコードを実行した際に出力を書き出し，それらの差分を取ることでリグレッションの原因を探ることが出来る．しかしこのデバッグ手法は print 文を埋め込む手間と書き出された出力の差分を人が解釈する必要があり手間と知識が要求される．

3　Bugbear の提案

本研究は元のプログラムのテストコードと変更後のプログラムのテストコードを実行した際の各関数呼び出しへの入出力の履歴を比較することでリグレッションの原因を特定するツール Bugbear を提案する．Bugbear は JavaScript で実装されており NodeJS で動く．バージョン管理システムとして git を採用しているプロジェクトでの利用を想定しており，比較したいプロジェクトの過去と現在のコミット ID を指定する．本ツールは 3 つの主要なコンポーネントによって

構成されている．(1) トレーサー (2) コンパレーター (3) ディテクターである．

3.1 トレーサー

トレーサーはプロジェクトを実行した際の各関数呼び出しの入出力として以下の情報の値を記録する．

- 関数名
- その関数呼びだしを呼び出している関数呼び出し
- 関数実行前の引数
- 関数実行前の this
- 戻り値
- 関数実行後の引数
- 関数実行後の this

this とは JavaScript のどのように関数が呼ばれたかによって決定される値[2] である．this はクラス内でインスタンスを参照するために利用されることがあり，Bugbear ではインスタンス変数にアクセスするために this を保存している．

実装にあたってソースコードを変換して元の関数を囲むトレース用関数を挿入するトレーサーを作成した．変換後は図 3 のように 2 行目のトレース用関数 `global.tracer.wrapCallback` が 3 行目から始まるコードブロックを囲んでいる．

トレーサーはトレースした変数の値を JSON 形式で保存する．実装にあたっては変数の値を JSON 形式に変換することが出来る `JSON.stringify` を利用した．例外的に循環参照が行われているオブジェクトは JSON.stringify で変換することが出来ないので循環参照を引き起こしているプロパティは ‘ -pruned- ’ という文字列に置き換えた．

全ての情報を保存するとログが膨大になるため性能に影響が出ない範囲で保存するデータの取捨選択をおこなう．具体的には this がグローバルオブジェクトやモジュールオブジェクトであるときは保存をしない．理由は大域変数を全て含み参照しないプロパティが多いと予想されることに加えてデータ量が多いからである．また関数呼び出しの ID 生成に使っている関数実行前の引数の値などは値が一致するかどうかだけわかれば良いのでハッシュ化しデータ量を削減している．

3.2 コンパレーター

コンパレーターは変更後のプログラムをトレースした結果、元のプログラムと比較して挙動が変わった関数をリグレッションの原因の候補とする．この際，元のプログラムのテストコードのトレース結果を保存しておき変更後のプログラムでテストコードを実行した際に動的にトレース結果が過去と変わっていないか比較する．

コンパレーターは同じ条件で呼び出されたにも関わらず違う振る舞いをするようになった関数をリグレッションの原因の候補とする．同じ条件で呼び出されているかを判別するために各関数呼び出しには呼び出しの条件によって定まる識別子を付与する．識別子は関数名，その関数を呼び出している関数呼び出し，関数実行前の引数，関数実行前の this のハッシュ値を連結して生成する．関数の振る舞いは戻り値，関数実行後の引数，関数実行後の this の中から実行時にツールに与える設定で決めた項目が一致すれば同一性の振る舞いをするとみなす．どの項目を振る舞いの判断に採用するかは設定で決められる他，Bugbear の機能で全ての設定を総当たりで調べることもできる．

コンパレーターの制限として，識別子が変わってしまう変更は追跡できない点がある．例えば関数名が変わると別の関数呼び出しとして扱われるため関数呼び出しの識別子が変わる．したがってコンパレーターでの比較がなされずリグレッションの原因特定の精度が下がってしまう．

3.3 ディテクター

ディテクターは関数の呼び出し関係を利用してリグレッションの大元の原因となった関数を絞り込む．

リグレッションが発生した際，一つの関数が原因であったとしても振る舞いが変わる関数は一つであるとは限らない．ある関数の振る舞いが変化すると，その関数を呼び出す関数も連動して振る舞いが変わる．しかし，リグレッションの原因である関数は大元の呼び出されている関数のみであると考えられる．

振る舞いが変わっている関数が全てリグレッションに関係しているとは限らない．意図した機能追加，バグ修正，リファクタリングなど，リグレッションでは

```
const sentenceConversion = () => {
  return global.tracer.wrapCallback("conversion.js:sentenceConversion", this, () => {
    const words = ['かん', 'が', 'いい'];
    return words.map(word => {
      return wordConversion(word);
    }).join("");
  })();
};
```

図 3　トレース用関数を挿入したソースコード

```
sentenceConversion('かんがいい');
```

図 4　実行コード

なくとも関数の振る舞いが変わることはある．リファクタリングの場合は最終的な結果が変化しないので実行したプログラムの結果が変わっていない場合はリグレッションとして扱わないと機械的に判断することが出来る．一方で意図した機能追加やバグ修正とリグレッションを機械的に区別することは出来ないため，Bugbear はプログラムの振る舞いが変わったこととその原因だけを提示し，その変更が意図したものであるかどうかは利用者が判断する．

3.4　利用例

本節では図 1 と図 2 に Bugbear を適用した例を紹介する．テストコードとして図 4 を実行したとする．関数の振る舞いを表す値には戻り値を使った．

図 5 は図 1 の `wordConversion` を引数「かん」で呼び出した際のトレース結果である．`id` は 3.1 で記述した方法で生成した関数呼び出しの識別子でハッシュ値の長さを 64 文字から 4 文字に省略して記述している．同じ関数呼び出しであることを判定する上で this や引数といった直接保存するとデータサイズが大きくなるものを利用している．それらを識別子にする際にそれらをハッシュ化することで大幅にデータサイズを削減出来ている．また複数の情報を一つの `id` の中に保持できているので同じ関数呼び出しの検索が関数呼び出しの件数に対して O(1) で出来るようになっていることもメリットである．`caller` はこの関数呼び出しを呼び出した関数呼び出しの識別子である．result はこの関数呼び出しの戻り値である．

```
{
  "id":"d56b:dca6:6853:----",
  "functionName":
    "conversion.js:wordConversion",
  "caller":"42db:7423:4f53:----",
  "result":"勘"
}
```

図 5　wordConversion のトレースデータ

```
{
  "null": [
    {
      "id": "42db:7423:4f53:----",
      "functionName":
        "conversion.js:sentenceConversion",
      "result": "勘がいい",
      "newResult": "勘, 缶がいい, 良い",
      "callees": [
        {
          "id": "d56b:dca6:6853:----",
          "functionName":
            "conversion.js:wordConversion",
          "result": "勘",
          "newResult": [
            "勘",
            "缶"
          ]
        },
        ...
        {
          "id": "d56b:dca6:f93b:----",
          "functionName":
            "conversion.js:wordConversion",
          "result": "いい",
          "newResult": [
            "いい",
            "良い"
          ]}]}]}
```

図 6　ツール適用結果

関数の呼び出し関係を考慮して挙動が変わったと判定される関数は図 6 に示す通りである．結果を見ると振る舞いが変わった関数は `sentenceConversion` と `wordConversion` であることがわかる．複数の関

数の振る舞いが変わった際に大元の原因となった関数はコールグラフ中の一番深い関数であると思われる．この場合，sentenceConversion の振る舞いが変わっているのは wordConversion の振る舞いの変化に連鎖的に影響を受けたことが理由である．つまり wordConversion が原因でリグレッションが発生していると判定する．

4 実験

本節では実施した実験について述べる．

4.1 実験の内容

リグレッションの原因を正しく検出できるかを実際のリグレッションを起こしているソースコードに本ツールを適用して確認した．対象とするプログラムとしてバグベンチマークスイートの BugsJS [5] からバグの種類がリグレッションであるものだけを抜き出して利用した．BugsJS は実際に Github で開発された 10 の OSS プロジェクト中からバグが発生しているコミットを選びラベルをつけたものである．本実験ではこの中の Hessian.js [1] というシリアライズツールを対象とした．Hessian.js はデータを Hessian という形式のバイナリ・ウェブサービス・プロトコルにシリアライズする．

BugsJS はリグレッションに特化したものではないため，我々はバグの中からリグレッションであるものだけを抽出した．バグを引き起こす前後のコミットではプログラムが正しく動いている場合，リグレッションであると判定する．バグを修正した後のコミットで作られたバグが直ったことを確認するためのテストを使い，プログラムが正しく動くかを判定する．バグを引き起こすコミットより前のコミットにそのテストを適用してテストが成功していた場合，そのバグは過去動いていたコードがバグにより動かなくなった，つまりリグレッションと判定する．

Hessian.js には 9 つのバグケースがあったがそのうち表 1 の 2 つがリグレッションであった．どちらのバグも if 文の条件式を変更することで修正されている．BugID2 は機能追加に伴いデータの key が null だった時の挙動が後方互換性を失っているというリグレッション，BugID9 は大きすぎる数字を number 型にエンコードすると情報が欠損するため string 型で返す仕様にしていたが，number 型を返してしまっているというリグレッションである．

```
1 describe('map.test.js', function() {
2   it('should ~~~', function() {
3     var data = new Buffer([77, 116, ~~~]);
4     var rv = hessian.decode(data);
5     rv.should.eql({null: 'null'});
6     ~~~~~~
7   });
8 });
```

図 7　BugID2：テストコード

```
1 AssertionError: expected Object {} to equal
  ↪  Object {null: 'null'}
2
3 at Assertion.fail (~~~/should.js:326:17)
4 at Assertion.value (~~~/should.js:398:19)
5 at Context.<anonymous> (~~~/map.test.js:168:15)
6 at process.processImmediate (~~~/timers:471:21)
```

図 8　BugID2：エラーメッセージ

Hessian.js には図 7 のようなテストコードが用意されている．テストコードは統合テストに近いもので実行するとテストが失敗し図 8 のエラーメッセージが表示される．このエラーメッセージからはどういった条件でリグレッションが発生するかまではわかっても，どの関数が原因であるかを判断するのは難しいだろう．

4.2 実験の結果

本ツールを用いると 2 つのリグレッションについてリグレッションの原因を正しく絞り込むことができた．BugID2 では変更されたファイルが 25 個で書き換えられた関数は 4 つ，振る舞いが変わった関数は 2 つ，という中から原因となっている関数を見つけることができた．BugID8 では変更されたファイルが 7 個で書き換えられた関数は 3 つ，振る舞いが変わった関数は 3 つ，という中から原因となっている関数を見つけることができた．このことからプログラムの変更が複数箇所で起きている場合でも機械的にリグレッションの原因を見つけることが出来たと言える．またリグレッションを直したコミットで変更された関数は本ツールが提示した関数と同じであった．本ツールが

BugID	変更されたファイル	書き換えられた関数	振る舞いが変わった関数	追加行数	削除された行数
2	25	4	2	187	87
8	7	3	3	27	14

表 1　リグレッションの実験ケース

提示する関数は実際の修正に役立つ情報であると考えられる．

5　関連研究

テストを自動で行うことを目的としてテストケース自動生成の研究が行われている．例として Evosuite [4] は高いカバレッジのテストコードを生成し現在の振る舞いの記述と将来の不具合の検出が出来る．本研究もテストを自動化するが，テストケース自動生成と比較するとバグの原因の特定までを自動化しようとしている点で異なる．

プログラムを実行した際の履歴を保存してデバッグに活用する研究として Time Travel Debugging (TTD) が挙げられる．TTD は実行中のプロセスの実行を全て記録し，後で実行を再現できるようにするツールである [3]．TTD はデバッグを支援するもので原因の特定自体はユーザーが行うが，本研究はツールが原因を特定することを狙っている．

6　まとめと今後の課題

本論文では過去と現在のプロジェクトのテストコードの実行履歴を比較してリグレッションが発生した原因となる関数を提案する手法について述べた．

今後は実験ケースを充実させること，Bugbear の出力を利用者に分かりやすくすること，仮説の検証を行う予定である．実験ケースについては対象とするバグケースを増やし検出精度の指標を用意して定量的な評価を行う．また出力については現在はリグレッションの原因の候補を json 形式で出力しているが，利用者がみた時にわかりやすいような形式で出力する．例えば同じ関数で振る舞いが変わった関数呼び出しは引数と出力を整理して提示するなどの工夫が考えられる．また本研究では複数の関数の振る舞いが変わった時，コールグラフ中の一番深いところにある関数が大元の原因であるという仮説の元でリグレッションの原因を絞り込んでいる．これは呼び出される側が呼び出した側に振る舞いの変化が連鎖することはあっても逆は起きないと考えているからである．今後はこの仮説の再評価を行う．

参 考 文 献

[1] ：hessian.js. https://github.com/node-modules/hessian.js.

[2] ：this - JavaScript — MDN. https://developer.mozilla.org/ja/docs/Web/.

[3] ： Time Travel Debugging - 概要 - Windows drivers. https://docs.microsoft.com/ja-jp/windows-hardware/drivers/debugger/time-travel-debugging-overview.

[4] Fraser, G. and Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, New York, NY, USA, Association for Computing Machinery, 2011, pp. 416–419.

[5] Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, c., Ferenc, R., and Mesbah, A.: BugsJS: a Benchmark of JavaScript Bugs, *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 90–101.