データ値の差異とデータフローの視覚化によるデバッグ補助手法の提案

神谷 年洋
島根大学大学院自然科学研究科
kamiya@cis.shimane-u.ac.jp

要旨

クラッシュしないバグのデバッグ，すなわち，プログログラムの実行中に不正な値が生成され，ヌルポインタエラーなどで中断されることなくプログラムの実行が続き，一見正常に終了するものの，不正な出力が得られる不具合を修正する状況を想定したデバッグの補助手法を提案する．

提案手法は動的解析手法の一種であり，対象となるプログラムを実行して実行トレースを収集し，実行トレースから不具合に関連するデータフローと関数呼び出しを含む部分列を抽出する．不具合に関連するデータフローを一覧できるようにすることで，開発者が不具合の原因を理解し，ソースコード上で修正を行うべき場所を検討する作業を補助する．抽出される実行トレースの部分列は，プログラムの実行開始から不具合に至った実行系列を含むものであるが，値の差異に着目した絞り込み，および，データフローによる絞り込みにより，もとの実行トレースよりも小さなものにすることで，一覧性を高める．

本提案手法におけるデータフロー追跡には，動的な解析を用いることにより，参照型（すなわち，ポインタで参照されるオブジェクトなど）のデータ以外にも，値型（整数やブーリアンなどの基本型を含む）のデータに関しても，データフローの追跡が可能になっている．値型のデータフローの追跡は，対象プログラムの実行プロセスを越えてデータが受け渡しされるものについても適用可能である．例えば，データベースやファイルにいったんデータを格納し，そのあと取り出すような場合でもデータフローを追跡することが可能である．

1. はじめに

ソフトウェアの不具合には，クラッシュするバグ (crashing bug) とクラッシュしないバグ (non-crashing bug) がある．クラッシュするバグとは，いわゆるヌルポインタの参照や配列の添字が範囲を超えていることにより，プログラムの実行が中断するものである．クラッシュするバグの場合には，クラッシュした実行時点でのスタックトレースを利用できる．スタックトレースには情報として，プログラムが中断した理由と該当するソースコードの場所（関数名やソースコードのファイル名，行番号など）に加えて，その関数を呼び出した関数，さらにその関数を呼び出した関数，などがエントリポイントまで順に含まれる．そのため，スタックトレースを手がかりとしてデバッグに着手することができる．それに対して，クラッシュしないバグでは，プログラムの実行中に不正な値が生成され，プログラムの実行状態中にその影響が伝播し，出力の一部が不正になる．プログラムが中断しないため，スタックトレースを手がかりとして利用することができない．

本稿では，クラッシュしないバグのデバッグを目的とするデバッグ補助手法を提案する．提案手法では，プログラムを実行して実行トレースを収集し，実行トレースから不具合に関連するデータフローと関数呼び出しを含む部分列を抽出する．実行トレースの部分列の視覚化を通じて不具合に関連するデータフローを一覧できるようにすることで，開発者が不具合の原因を理解しソースコード上で修正を行うべき場所を検討する作業を補助する．

以降，2 で関連する研究について説明する．3 ではトイプログラムを対象として手法の利用方法を説明する．4 で提案手法について説明し，5 では，提案手法を評価するための予備的な実験として，あるオープンソースプ

ロダクトと先のトイプログラムへの適用について，実行時性能も含めた評価を行う．最後に，6 でまとめと展望について述べる．

2. 関連研究

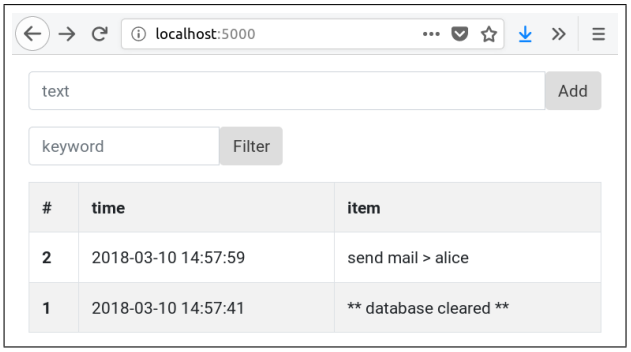
クラッシュしないバグのデバッグを補助するための手法が種々提案されている．

プログラムスライシング [15]，特に，バックワードスライシングとは，不具合（典型的にはプログラムが出力した不正な値）を格納した変数の値を計算するのに利用された文を抽出する手法である．プログラム中の文や式の間に，ある文や式を実行することを決定した文の関係（制御依存）や，ある変数の値を計算するために利用された変数の関係（データ依存関係）をプログラム依存グラフ (PDG) と呼ばれるグラフとして表現する．不具合が再現した（不正な値が代入された）変数をスライシング基準（起点）としてグラフをたどることで，不具合の原因となり得るプログラムの一部分を特定する．プログラムスライシングには様々な変種が提案されていて [14]，近年に至るまで技術的な改良が続けられている [3]．また，不具合の位置を特定する精度についても実験的な評価が行われている [12]．

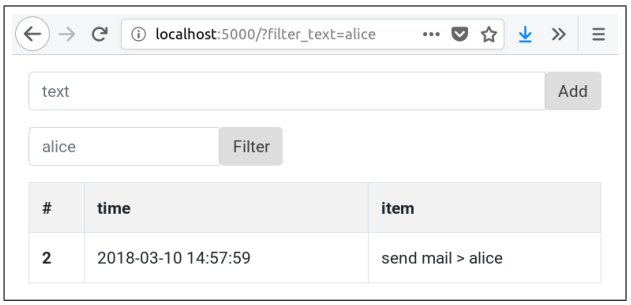
デルタデバッギングと呼ばれる手法には，プログラムの異なるバージョン（リビジョン）を同じ入力に対して実行し，不具合が再現するバージョンと再現しないバージョンを特定し，両者のソースコードの差分に不具合の原因があると推定する手法 [16] と，入力データを少しづつ変更していき，不具合を再現する最小の入力を生成する手法 [17] がある．

スペクトラムベース，あるいは，統計的デバッグと呼ばれる手法 [2][4] では，まず，対象となるプログラムに対する入力（テストケースなど）を多数用意する．それらの入力を与えてプログラムを実行し，プログラム中の文のそれぞれが実行されたか否かを記録しておく．不具合を起こした実行において，特に多く実行された文を不具合の原因であると推定する．

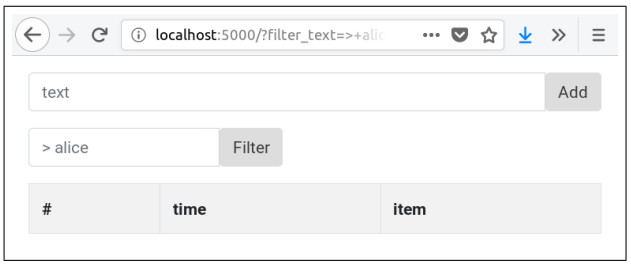
逆戻りデバッグ，あるいはキャプチャ＆リプレイと呼ばれる手法は，プログラムの実行トレースを記録しておくことで，従来のデバッガのステップ実行とは異なり，プログラムの実行時の任意の時点からプログラムの実行の順向きあるいは逆向きにステップ実行する（実行したかのように再現する）ことを可能にする [4][13]．



(a) メモ項目「send mail > alice」を追加した直後



(b) フィルタリング文字列「alice」を指定



(c) フィルタリング文字列「> alice」を指定（不具合）

図 1. メモアプリ実行例

データフローやデータの参照を主要な対象とする動的解析手法も提案されている．オブジェクトの参照を依存関係と捉え，ソフトウェアの機能間の依存関係を特定するオブジェクトフロー分析 [9]，不正なデータフローを検出する動的情報流解析 [10] が提案されている．近年では，静的解析と動的解析を併用し，依存関係を多段に辿って解析を進めていくハイブリッドなバグ位置特定の手法 [1][11] も提案されている．

3. トイプログラムへの適用例

あるトイプログラムに提案手法およびその実装を適用した例を示す．

```
@app.route("/")    ← トップページ
def index_page():
    cur = get_db(g).cursor()
    cur.execute("SELECT id, updated, item FROM memo ORDER BY updated DESC;")
    records = cur.fetchall()                                  } 記録日時の降順にデータベースからメモ項目を取得する

    filter_text = request.args.get('filter_text', None)
    if filter_text:                                           } フィルタリング文字列が設定されている場合、文字列を含むメモ項目のみを残す
        records = [r for r in records if filter_text in r[2]]

    …フォーマットして出力する（省略）…

@app.route("/add", methods=['POST'])    ← メモ項目の追加ボタンの処理
def add_request():
    item_text = request.form['item']
    item_text = bleach.clean(item_text.strip())               } フォームに記入されたメモ項目の文字列を取得し、サニタイズする

    if item_text:
        sql = "INSERT INTO memo (updated, item) VALUES (?, ?);"
        get_db(g).cursor().execute(sql, [datetime.now(), item_text])   } 空の文字列ではない場合には、データベースにメモ項目を追加する

    return redirect('/')    ← トップページにリダイレクト（してメモ項目を表示）する

@app.route("/filter", methods=['POST'])    ← フィルタリングボタンの処理
def filter_request():
    filter_text = request.form['filter']    ← フィルタリング文字列を取得する

    if filter_text:
        param_str = '?' + urllib.parse.urlencode({'filter_text': filter_text})
        return redirect('/' + param_str)                      } フィルタリング文字列を設定してトップページにリダイレクトする
    else:
        return redirect('/')
```

図 2. メモアプリのソースコード（抜粋）

3.1. 対象プログラム

対象となるプログラムはメモを書き留めるための web アプリケーションであり，web フレームワークとデータベースを再利用して実装されている．ユーザーがメモ項目を記入する機能，記入したメモ項目を一覧する機能，フィルタリング文字列を指定してその文字列が含まれるメモだけを一覧する機能を持つ．このフィルタリング機能には意図的に不具合が作り込まれている．図 1 の (a) から (c) に対象プログラムの画面のスクリーンショットを示す．(b) はフィルタリング機能が期待通りに動作している例，(c) はフィルタリング機能が不具合を起こしている例である．(c) では，フィルタリング文字列「> alice」を含むメモ項目を表示するように指定しているが，その文字列を含む「send mail > alice」というメモ項目が表示されていない．

図 2 に，対象プログラムのソースコードの一部を示す[1]．関数 `index_page` はメモ項目を一覧するページがリクエストされたとき，ページを生成して返す関数である．フィルタリング文字列が設定されていれば，メモ項目を選別してからページの生成を行う．関数 `add_request` はメモ項目を追加するボタン（画面の「Add」）が押されたときのリクエストを処理する関数である．入力されたテキストを新たなメモ項目としてデータベースに格納する．関数 `filter_request` はフィルタリングボタン（画

[1]対象プログラムのソースコード全体は `https://github.com/tos-kamiya/memo.py` を参照のこと．

```
def test_filter_entries_buggy(app):
    # メモ項目「send mail > alice」を追加する
    rv = app.post('/add', data=dict(
        item="send mail > alice"
    ), follow_redirects=True)

    # フィルタリング文字列「> alice」を設定する
    rv = app.post('/filter', data=dict(
        filter="> alice"
    ), follow_redirects=True)

    # フィルタリングの結果を取得する
    print(repr(extract_content_of_id(
        rv.data, 'items')))
```

図 3. メモアプリの不具合を再現するスクリプト (抜粋)

```
        1-1{    _:1 __main__//test_filter_entries_buggy
        1-2     t.py:17 .load
!2#2007 1-2:2   'send mail > alice'
        1-3-1{  t.py:19 werkzeug.test/FlaskClient/post
(snip)
!1#1010 1-4:1   'Ialice' '>␣alice'
        1-5-1{  t.py:22 werkzeug.test/FlaskClient/post
(snip)
        1-5-3-4-6-6-4-4-4-4-2-5-5-4-1{  L/flask/app.py:1598 memo//index_page
(snip)
        1-5-3-4-6-6-4-4-4-4-2-5-5-4-3-4-3-3-3-2-4-2-3}  L/werkzeug/urls.py:536 .ret
!1#1010 1-5-3-4-6-6-4-4-4-4-2-5-5-4-3-4-3-3-3-2-4-2-3:1 'Ialice' '>␣alice'
(snip)
        1-5-3-4-6-6-4-4-4-4-2-5-5-4-7}  T/memo.py:40 .ret
!1!2#1008  1-5-3-4-6-6-4-4-4-4-2-5-5-4-7:1 '⏭"Ialice⏭\n<tr><th>2</th><td>2018-03-11␣04:33:44</td⏭\n⏭
'⏭">␣alice⏭\nI\n⏭'
```

図 4. メモアプリのデータフローの視覚化（抜粋）

面の「Filter」）が押されたときのリクエストを処理する関数である．入力されたフィルタリング文字列をパラメータとして設定しつつメモ項目を一覧するページにリダイレクトする．

3.2. 不具合の原因

上述のフィルタリング機能の不具合の原因は，ユーザーが入力した文字列をサニタイズ（特殊な意味を持つ文字「&」や「>」などをエスケープ）する処理としない処理が混在するためにデータの一貫性がなくなることである．関数 `add_request` ではメモ項目の文字列をサニタイズしている（関数 `bleach.clean` を利用）のに対して，関数 `fiter_request` ではフィルタリング文字列をサニタイズしていない．結果として，関数 `index_page` でメモ項目をフィルタリングする処理ではサニタイズ済みの文字列の中から，サニタイズされていない文字列を検索することになる．不具合が再現する図 1 の (c) の実行例では，フィルタリング文字列にサニタイズによって置き換えられる文字「>」が含まれているため，検索結果が空になっている．

この不具合は，いわゆるクラッシュしないバグであり，プログラムは動作し続けるが出力は不正なものとなる．さらに，この例では，不具合の原因に関係する 3 つの関数の個々について単体では処理の誤りを議論することはできず，3 つの関数の仕様や処理，データフローを理解して突き合わせることが必要となる．不具合の修正としてフィルタリング文字列をサニタイズするよう処理を修正することにした場合には，修正される関数は `fiter_request` と `index_page` のいずれかとなる．

3.3. 提案するデバッグ補助手法の手順と適用例

提案手法では，対象プログラムを（不具合が再現する入力データを始めとして）いくつかの入力データを与えて実行し，取得した実行トレースからデータフローを特定し (4.1 で後述)，その後，データ値やデータフローにより実行トレースを絞り込み視覚化する (4.2 で後述) ことで分析を進める．ユーザー（開発者）は不具合に関係がありそうなデータ値とデータフローにより実行トレース内で絞り込みを行い，不具合の原因となり得る関数の候補を選びだし，そのソースコードを参照して不具合の原因であるかを確認する．

図 3 に，不具合を再現する入力を与えてメモアプリのプログラムを実行するスクリプトの一部を示す（このスクリプト，および，このスクリプトを修正したものを用意し，複数の実行トレースを取得して，提案手法の入力とした）．

図 4 に，提案手法を実装したツールによる解析結果を示す．実行トレースから差分データを含むデータフローを抽出し視覚化したものの一部である (抽出された関数呼び出しは 267 回と一見多く見えるが，web フレームワークによる関数間でのデータのとりまわしが大量に含まれており，ユニークなデータ値は 62 種類である．表 2 を参照)．各行に，データフローを識別するラベル（「!1」や「#1010」など．詳細は 4.1 で後述），コールツリー内での位置，呼び出された関数の名前（パッケージ名，メソッドの場合はクラス名も含む）や，実行されているソースコードの行（ファイル名と行番号），あるいは，引数や戻り値として現れたデータ値が示されている．

図中で「!2」が付加された行は，メモアプリに入力し

たメモ項目のデータに関するデータフローを示している．メモ項目のデータはデータベースを介してやり取りされる（一度データベースに格納され，その後表示のためにデータベースから取り出される）が，そのようなデータに対しても，提案手法ではデータフローを追跡することができている．

図中で「!1」が付加された行は，メモアプリに入力したフィルタリング文字列のデータに関するデータフローを示している．関数 index_page に，URL にエンコードされたパラメータとしてフィルタリング文字列が渡され（図中「!1#1010」が付加された行），検索結果の画面の HTML 記述が生成されている（図中「!1!2#1008」が付加された行）．さらに，正しい検索結果（「<tr>」で始まる文字列）と不具合である空の検索結果の差分も示されている．この関数 index_page は，不具合として現れるデータを生成した処理を含み，（前述のように）不具合の修正のために変更される関数の候補でもある．

4. 提案するデバッグ補助手法

提案するデータ値の差異とデータフローの視覚化によるデバッグ補助手法は，基本的には，データ値の差異に基づくデータフロー解析手法 [8] と，実行トレースの検索のための技術 [6], [7] を組み合わせたものであり，特にデバッグを補助するために応用したものである．

4.1. データ値の差異に基づくデータフロー解析

対象プログラムに対して，データ（入力や出力，変数）の値が異なる（**変異**させた）2 つの実行トレースを用意し，それらの実行トレース中の対応する位置に現れるデータ（関数の実引数や戻り値として）の値の「差異を調べる」ことで，プログラムの実行中にシステムの中でそのデータがどのように伝播していったかを特定する解析手法である．

(1) 複数の実行トレースから同じ関数呼び出し列を抽出する前処理

異なる入力を与えて実行したときの実行トレースの間では，一般に，呼び出される関数や呼び出しの順序が異なる．実行トレース中の対応する位置にに現れるデータ値を比較できるようにするために，呼び出される関数や呼び出しの順序が異なる部分を取り除く前処理を行う．この前処理は，与えられた 2 つの実行トレースをコールツリーに変換し，それらコールツリーの根から順に幅優先で節点同士を比較していき，異なる関数呼び出しが現れたら，その節点から葉まで（その節点を根とする部分木）を相違点であるとみなして取り除く．この前処理により，呼び出されている関数列（どの関数がどの順序で呼び出されているか）は全く同じで，引数や戻り値として渡されるデータ値のみが異なる 2 つの実行トレースを生成する．

(2) データ値の差異からのラベルの生成

2 つの実行トレースの間の差異を表すラベルとして「**変異効果ラベル**」と「**値ペアラベル**」の 2 種類がある [8]．2 つの実行トレースの間で，同じ位置にあるデータ（実行トレース間で対応する関数呼び出しの同じ引数）の値が異なるときに，それらデータ値の違いは実行トレースを生成するのに用いられた入力データの変異に起因することを意味する変異効果ラベルを付与する．図 4 の例では，左端にある「!1」や「!2」が変異効果ラベルである．

変異効果ラベルは変異（すなわち入力データ値の差異）ごとに 1 種類づつ生成されるため，より詳細にデータフローを区別するために値ペアラベルを導入する．変異効果ラベルが付与されたデータ値のそれぞれについて，2 つの実行トレース内の対応する位置のデータ値のペアを生成し，値のペアが等しい部分には同じ値ペアラベルを付与する．例えば，一方の実行トレース内で変異効果ラベルが付与されたデータ値が順に $1, 2, 1, 1$ であり，もう一方の対応する実行トレース内でのデータ値が順に $10, 20, 11, 10$ であるとき，これらのデータ値から順に $(1, 10), (2, 20), (1, 11), (1, 10)$ という値ペアが生成される．前者の実行トレース中で現れる 3 つのデータ値 1 のうち，1 番目と 3 番目には値ペア $(1, 10)$ に相当する値ペアラベルが付与されるが，2 番目の 1 には異なる値ペア $(1, 11)$ に相当する値ペアラベルが付与され，従って，2 番目のデータ値 1 は 1 番目や 3 番目のデータ値とは異なるデータフローにあると判定される．図 4 の例では，「#1008」,「#1010」,「#2007」が値ペアラベルである．

4.2. データ値の差異を利用したデータフローの視覚化

実行トレース内で呼び出しが起きた順に関数呼び出し（あるいは関数呼び出しからのリターン）を並べて表示し，呼び出しの深さに応じてインデントする（図 4 はこ

の視覚化の例になっている）．このとき，前述の変異効果ラベルや値ペアラベルを指定してフィルタリングを行うことで，着目しているデータ値に関連するデータフローのみを絞り込んで表示することができる．

実行トレースをフィルタリングする手順は次の通りである．指定されたラベルの集合を L として，

(a) ある関数の呼び出し c の引数や戻り値のデータ値にラベルが付与されていて L の要素であるる場合にはその関数の呼び出し c とそのリターン，引数や戻り値のデータ値を表示する．

(b) ある関数の呼び出し c の中で，直接または間接的に別の関数を呼び出し d を行っていて，d が上の (a) で表示される場合には，c やそのリターンを表示する．

提案手法を実装したツールでは，ユーザーがラベルを指定してフィルタリングすることにより，より少数のデータフローや関数呼び出しのみに着目して分析を進める（フィルタリングの利用例は 5.3 で後述）．

4.3. 変異させた入力データの準備

提案手法では，対象プログラムに対して異なった入力データを与えて実行することにより収集した複数の実行トレースが必要となる．不具合を再現する入力データを「少しだけ」（2 つの実行トレースの間で，なるべく，条件分岐の同じ枝が実行されるように）変更する（変異させる）ことで，複数の入力データを用意する．実行トレース間で呼び出される関数が異なっている部分は 4.1 の (1) の処理の対象となりデータフローの検出が行えないため，そのような部分を減らすことが望ましい．入力データを変異させる作業はユーザー（開発者）に任されている．その理由は，提案手法において，不具合に関係しつつ呼び出される関数に影響を与えないような入力データの変異を作成するためには，対象プログラムの処理内容やそのドメイン，不具合の内容に関する知識からの判断が必要となり，現状では自動的に行うことが難しいためである．

ただし，入力データの変異が小さいものであるほど，実行トレース中にデータ値の差異が現れる箇所は少なくなり，結果的に，特定できるデータフローも少なくなってしまうことが想定される．なるべく多くのデータフローを抽出するため，変異させた入力データを多数用意して，その結果をマージすることが可能になるツールを実装に含めた．これにより，実行トレース中のより広範囲にデータ値の差異を出現させることで，より広範囲にデータフローを抽出できるようにしている．例えば，3 の適用例では，メモ項目の文字列を変異させたものと，フィルタリング文字列を変異させたもの，2 つの変異を利用している（具体的な変異の内容については 5.2 で後述）．

5. 予備的な実験

提案手法を性質の異なる 2 つのプログラムに適用した例を示す．ひとつは 3 で前述したメモアプリに意図的に作り込まれている不具合を提案手法により特定できるか確認するための適用例であり，もうひとつは，オープンソースのライブラリの実際のデバッグに適用した例である．これらの実験では 3.3 の冒頭で示した手順を何度か繰り返して行うことにより不具合の原因とみなせるソースコードの記述を特定した．

5.1. 実装および実行環境

提案手法を実装したツールは，Python で記述されたプログラムを解析の対象とし，ツール自体も Python で記述されている．対象プログラムを実行し実行トレースを取得するツールは 1,574 行，実行トレースを解析しデータ値の差分からデータフローを抽出し視覚化するツールは 2,426 行である．

ツールを実行した PC は，CPU Intel Core i7-6800K CPU 3.40GHz，RAM 128GiB を備える．ツールはシングルスレッドで実行したため，以降で示すツールの実行時間（経過時間）は，ほぼそのまま CPU 時間となっている．対象プログラムの実行（および実行トレースを取得するツール）を実行するのに利用した VM（仮想機械，インタプリタ）は CPython 3.6.3 (with GCC 7.2.0) であり，データフローを抽出し視覚化するツールの実行に利用した VM は Pypy 3.6.3 (PyPy 5.10.1 with GCC 6.2.0) である．

5.2. メモアプリへの適用例

3 で導入したメモアプリは，提案手法の説明のために作成されたトイプログラムであり，ソースコードは主として Python で記述され，5 つの関数，全体で 102 行というごく小規模なものであるが，データベースや web フレームワークといったライブラリを利用しているため，

プログラムを実行すると多くの関数呼び出しが行われる．不具合の特定のためには，それらのライブラリの仕様や動作も含めて調査していく必要がある．

セットアップ

実行トレースを収集するために，対象プログラムの入力データとして，不具合を再現する入力データと，それを次のように変異させた入力データを準備した．

- **変異 m vs M** データベースに格納するメモ項目として「send mail > alice」，対，一部を大文字に変えた「send MAIL > alice」
- **変異 a vs g** フィルタリング文字列として「alice」，対,「> alice」

変異`m vs M`はメモ項目として入力されたデータのデータフローを追跡するため，変異`a vs g`はフィルタリング文字列のデータフローを追跡するとともに，不具合が再現する実行トレースと再現しない実行トレースを作るためのものである．

解析結果

変異`m+a`，`m+g`，`M+a`の組み合わせのそれぞれに対して対象プログラムを実行する（web アプリのサーバーにリクエストを送り，そのレスポンスを確認する）単体テストプログラムを作成して利用した．表 1 に実行トレースの取得とデータフローの抽出に要した時間やピーク（レジデント）メモリ消費量を示す．

表 2 に取得した実行トレースの大きさ，および，抽出されたデータフローを含む（すなわち，抽出された異効果ラベルや値ペアラベルのいずれかを含むという条件でフィルタリングした）実行トレースの部分列の大きさを示す．実行トレース全体ではのべ 21 万回以上の関数呼び出しがあり，モジュールのローディングのための処理などを取り除いた後の，エントリポイント以降の部分列でものべ 6 千回以上の関数呼び出しがある．提案手法により，ラベルが付加されたデータ値を含む関数呼び出しに着目することで，267 回にまで絞り込めたことがわかる．

この実験では，データフローを示すラベルのうち，基本型のデータ値に付加されたラベルのみを利用し，オブジェクトに付加されたラベルは利用しなかった（web アプリケーションであるため通信路でデータが変換が行われる，データベースへのデータの格納・取得に伴い異なるオブジェクトに変換される，などの理由により，オブジェクトによりデータフローを追跡するのは手間がかかるため）．表 2 でも，オブジェクトに付加されたラベルは数字に入れていない．

表 1. メモアプリの解析に要した実行時間およびメモリ消費量

ステップ	実行時間 (秒)	メモリ消費量 (KiB)
実行トレースの取得 (1 回あたり，最大)	101	44,220
データフローの抽出	60	126,496

5.3. DeepDiff への適用例

対象となったプロダクトである DeepDiff(`https://github.com/seperman/deepdiff/`) はオープンソースのライブラリであり，Python の種々のデータ，特にリストやタプルなどの構造を持つデータの差分を抽出する機能を持つ．ソースコードの規模はプロダクションコード 2,852 行，テストコード 2,716 行である．

5.2 のメモアプリと比較すると，DeepDiff はそれ自身が複雑なロジック（データ構造を再帰的に探索して差分を計算する）を持ち，再利用するライブラリが少ない（小さい）．また，データベースや web フレームワークなどのデータを変換するライブラリを利用していないため，オブジェクトに付加されたラベルをデータフローの追跡のために有効に利用できると想定される．また，不具合は実験のために作り込まれたものではなく，github 上ではプロジェクトの issue として報告されている，論文投稿時点で未解決のものである．

セットアップ

不具合の内容は，NumPy（数値演算ライブラリ）の配列の差分を DeepDiff で正しく求められないことがあるというものである（`https://github.com/seperman/`

表 2. メモアプリからのデータフローの抽出

実行トレース	関数呼び出し (回数)
m+g	212,601
エントリポイント以降	6,827
データフロー (ユニークなデータ値)	267 (62)

```
import deepdiff
import numpy as np

def main():
    d1 = np.array([2, 3], dtype=int)
    d2 = np.array([2, 3], dtype=int)
    ans = deepdiff.DeepDiff(d1, d2, verbose_level=2)
    print(repr(ans))

if __name__ == '__main__':
    main()
```

図 5. DeepDiff の不具合を再現するスクリプト

deepdiff/issues/97)．不具合を再現するスクリプトを図 5 に示す．このスクリプトは，要素の型として int を指定して生成した NumPy の配列 (numpy.array)2 つを比較する処理を行うものであり，全く同じ内容の配列を比較しているために「差分がない」という出力が期待されるが，実行すると「データが処理されなかった」という出力が得られた．このスクリプトの「np.array([2, 3], dtype=int)」の部分 2 箇所を，「np.array([2.0, 3.0], dtype=float)」に変更したものは，実行すると期待通りの「差分がない」という出力が得られるため，不具合の原因は「DeepDiff が NumPy の配列に未対応である」といった単純なものではないと推定された．

対象プログラムの入力データとして，不具合を再現する図 5 のスクリプト（**不具合再現スクリプト**とする）と，前述の int を float に変異させたもの（**変異スクリプト**とする）を利用した．

解析結果

不具合再現スクリプトと変異スクリプトを実行して取得した 2 つの実行トレースを入力として解析を行った．表 3 に実行トレースの取得とデータフローの抽出に要した時間やピーク（レジデント）メモリ消費量を示す．

ソースコード内の不具合の原因だと判断した部分に至るまでに，表示するラベルの範囲を変更してフィルタリングを行うことにより，計 3 回データフローの視覚化を行うこととなった．表 4 にこれら 3 回の視覚化によって得られたデータフローの部分列の大きさを示す．最初に視覚化したデータフロー (1) は 5.2 と同様に基本型のデータ値に付加されたラベルのみを利用したものである．この視覚化の結果では，対象プログラムの入力データ値の違い（int 型と float 型の違い）や，出力されるメッセージの違い（「差分がない」と「データが処理されなかった」）は確認できたが，差分の計算に相当する処理はデータフローには含まれなかった．次に，出力されるメッセージの生成に用いられたデータ値を確認するために，データフロー (2) として，基本型のデータのみではなくオブジェクトに付加されたラベルも含むデータフローの視覚化を行った．このデータフローを含む実行トレースの部分列は 203 回の関数呼び出しを含むが，分析に当たって参照したものは，メッセージの生成の直前のデータ値のみである．このデータ値 (後述の図 6 ではオブジェクト ID 7fce77d22b48) は「deepdiff.diff/DeepDiff」という型を持っていたため，対象プログラムの処理に深く関係していることが期待された．3 度目に，基本型のデータ値に付与されたラベルとこのオブジェクト 7fce77d22b48 に付与されていた値ペアラベル「#150」を含むデータフローの視覚化を行ったものは，データフロー (3) として示される 13 回の関数呼び出しを含んでいた．図 6 に視覚化の結果を示す．この図に含まれる関数呼び出しのうち，オブジェクト 7fce77d22b48 が 3 回目に利用されている箇所（メソッド__skip_this）の処理を確認したところ，差分計算の対象となるデータを型により識別するロジックが含まれていた．図 5 の不具合再現スクリプトおよび変異スクリプトで利用されているデータ型を（スクリプトに「print(type(d1[0]))」などの文を追加して実行することにより）確認したところ，配列の生成の直後に要素の型はそれぞれ numpy.int64 と numpy.float64 に

表 3. DeepDiff の解析に要した実行時間およびメモリ消費量

ステップ	実行時間 (秒)	メモリ消費量 (KiB)
実行トレースの取得 (1 回あたり，最大)	11	32,548
データフローの抽出	3.5	79,976

表 4. DeepDiff からのデータフローの抽出

実行トレース	関数呼び出し (回数)
不具合再現スクリプト	29,728
エントリポイント以降	548
データフロー (1)	8
データフロー (2)	203
データフロー (3)	13

```
                1-1{          _:1  __main__//main
                1-2           ddbad-target.py:5  .load
!1              1-2:1         2       2.0
!1              1-2:2         3       3.0
                1-3-1{        ddbad-target.py:7  DeepDiff/__init__
!1!2#150#2138   1-3-1:1       <DeepDiff 7fce77d22b48>  <DeepDiff 7f602eb48af0>
                1-3-4-1{      L/deepdiff/diff.py:646  deepdiff.diff/DeepDiff/__diff
!1!2#150#2138   1-3-4-1:1     ↑       ↑
                1-3-4-2-1{    L/deepdiff/diff.py:1073 deepdiff.diff/DeepDiff/__skip_this
!1!2#150#2138   1-3-4-2-1:1   ↑       ↑
(snip)
                1-4-1{        ddbad-target.py:8  agm_trace.hook//_repr
!1!2#150#2138   1-4-1:1       ↑       ↑
                1-4-2}        L/agm_trace/hook.py:126 .ret
!1#101          1-4-2:1       "{I'unprocessed':␣['root[1]:␣3␣and␣3',␣'root[0]:␣2␣▸▸  '{}'I
                1-5-1{        ddbad-target.py:8  agm_trace.hook/Out/write
!1#101          1-5-1:2       ↑       ↑
```

この図で「↑」とある部分は，直前と同じデータ値が出現していることを示す．

図 6. DeepDiff のデータフロー (3) の視覚化（抜粋）

```
>>> import numpy as np

>>> isinstance(np.int64(2), int)
False

>>> isinstance(np.float64(2.0), float)
True
```

図 7. NumPy の独自の型のデータと基本型との関係

なっていることが判明した．さらに，これらの型と上述のロジックで比較対象として記述されている型との関係を調査した（その一部を図 7 に示す）．

これらの調査から，不具合の原因として次が推定された．

- 不具合再現スクリプトおよび変異スクリプトにおいて，配列の要素は配列が生成された時点で NumPy ライブラリ独自の型（`numpy.int64` および `numpy.float64`）になっている．上述のロジックはこれらの型に関する記述を含まないため，これらの型自体は差分計算の対象ではない．
- ただし，`numpy.float64` 型のデータは対応する基本型 `float` のインスタンスであるとみなされる（それに対して，`numpy.int64` と基本型 `int` との間には関係は見られない）．`float` 型は上述のロジックにおいて差分計算の対象であるため，結果として，変異スクリプトでは差分が計算された．

この推定が正しいかを確認するために，上述のロジックのソースコードを修正して `numpy.int64` を差分計算の対象に含めるようにしたところ，不具合再現スクリプトの出力が変化し不具合が再現しなくなった．

6. まとめと展望

クラッシュしないバグのデバッグを目的とした動的解析手法として，実行トレース間のデータ値の差分に注目したデータフローの抽出手法を提案した．提案手法を実装したツールの予備的な適用実験を行い，あるオープンソースプロダクトのある既知（であるが未修正の）不具合に適用し，その原因を特定した．

提案手法は研究の初期段階にあり，適用対象のスケーラビリティやデータフロー抽出の精度の評価を含めて，より大きなプロダクトを対象とした実験による評価が必要である．また，提案手法の適用性や分析作業の容易さを改善するために，次のような課題に取り組む必要がある．

- 解析対象プログラムの入力（変異された入力データ）をユーザー（開発者）が手作業で準備する必要があるため，どのような変異を行うべきかの知見を積み上げる（4.3 を参照）．
- 解析対象プログラムの入力として単体テストを流用することを想定し，デバッグに有効なデータフローを抽出できる単体テストを選択する手法を提案する．
- 変異ペアラベルや値ペアラベルが多くなった場合の分析作業を容易にするため，ラベル間の関係（階層

性など）に基づいてラベルを整理したり追跡したりするためのツールを提供する．

謝辞

本研究は JSPS 科研費 16K12412 の助成を受けたものである．

参考文献

[1] Baah, G., Podgurski, A., Harrold, M., “The Probabilistic Program Dependence Graph and its Application to Fault Diagnosis”, *IEEE Trans. Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[2] Dallmeier, V., Lindig, C., Zeller, A., “Lightweight Defect Localization for Java”, *Proc. the 19th Annual Meeting of the European Conference on Object-Oriented Programming (ECOOP ’05), LNCS 3568*, Springer-Verlag, pp. 528-550, July, 2005.

[3] 石尾, 仁井谷, 井上, “プログラムスライシングを用いた機能的関心事の抽出”, コンピュータソフトウェア, 26 巻, 2 号 pp. 2_127–2_146, 2009.

[4] Jones, J.A., Harrold, M.J., “Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique”, *Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering* (ASE ’05), pp. 273-282, 2005.

[5] Joshi, S., Orso, A., “SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions”, *Proc. the 23ed IEEE International Conference on Software Maintenance* (ICSM ’07), pp. 234–243, 2007.

[6] 神谷 年洋, “And/Or/Call グラフの提案とソースコード検索への応用”, 電子情報通信学会技術研究報告, vol. 113, no. 269, pp. 173–178, 2013.

[7] 神谷 年洋, “逆戻りデバッグ補助のための嵌入的スパイの試作”, 電子情報通信学会技術研究報告, vol. 116, no. 127, SS2016-9, pp. 87–92, 2016.

[8] 神谷 年洋, “実行トレース間のデータの差異に基づくデータフロー解析ツール”, 電子情報通信学会技術研究報告, vol. 116, no. 136, pp. 55–60, 2017.

[9] Lienhard, A., Greevy, O., Nierstrasz, O., “Tracking Objects to Detect Feature Dependencies”, *Proc. the 15th International Conference on Program Comprehension* (ICPC ’07), pp. 59–68, 2007.

[10] Masri, W., Podgurski, A., “Algorithms and Tool Support for Dynamic Information Flow Analysis”, *Information and Software Technology*, vol. 51, no. 2, pp. 385–404, 2009.

[11] 中野, 大沼, 小林, 石尾, “動的データ依存集合の発生確率を用いた欠陥箇所特定支援手法の実装及び評価”, 電子情報通信学会技術研究報告, vol. 114, no 510, pp. 19–24, 2015-03.

[12] 西松, 西江, 楠本, 井上, “フォールト位置特定におけるプログラムスライスの実験的評価”, 電子情報通信学会論文誌 D, vol. J82-D1, no. 11, pp. 1336–1344, 1999.

[13] 櫻井, 増原, 古宮, “Traceglasses : 欠陥の効率良い発見手法を実現するトレースに基づくデバッガ”, 情報処理学会論文誌プログラミング (PRO), 3 巻, 3 号, pp. 1–17, 2010.

[14] Tip, F., “A Survey of Program Slicing Techniques”, *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[15] Weiser, M., “Program Slicing”, *IEEE Trans. Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[16] Zeller, A., “Yesterday, My Program worked. Today, It Does Not. Why?”, *Proc. the 7th European Software Engineering Conference* (ESEC/FSE-7), pp. 253–267, 1999.

[17] Zeller, A., Hildebrandt, R, “Simplifying and Isolating Failure-Inducing Input”, *IEEE Trans. Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.