情報処理学会論文誌　Vol.59　No.4　1137–1149 (Apr. 2018)

Javaプログラムのデータ遷移可視化ツールTFVISの開発

佐藤 拓弥[1]　片山 徹郎[1,a)]　喜多 義弘[2]　山場 久昭[1]　油田 健太郎[1]　岡崎 直宣[1]

受付日 2017年8月2日, 採録日 2018年1月15日

概要：ソフトウェア開発におけるデバッグは，手間のかかる工程である．プログラムの欠陥を効率良く特定するためには，プログラム実行時の挙動を把握することが重要である．しかし，プログラム実行時の挙動は一般的に不可視であり，その挙動がどこまで正しかったのかを把握することは困難である．そこで，本研究では，Java プログラムのデバッグの効率化を目的として，プログラムの欠陥特定を支援するデータ遷移可視化ツール TFVIS（Transitions and Flow VISualization）を開発した．TFVIS は，プログラム実行時のデータ遷移と実行フローの可視化を行う．評価実験では，TFVIS を活用することで欠陥特定に要する時間を約 35%削減できることを確認した．このことから，TFVIS による可視化がデバッグ効率の向上に効果的であるといえる．

キーワード：デバッグ，可視化ツール，動的解析，データ遷移，Java

Development of TFVIS (Transitions and Flow VISalization) for Java Programs

Takuya Sato[1]　Tetsuro Katayama[1,a)]　Yoshihiro Kita[2]　Hisaaki Yamaba[1]　Kentaro Aburada[1]　Naonobu Okazaki[1]

Received: August 2, 2017, Accepted: January 15, 2018

Abstract: It takes much time in debugging process. To find bugs effectively, it's important to understand the dynamic behavior of programs. But it's difficult since the dynamic behavior of the program is generally invisible. To support understanding the dynamic behavior, we have developed TFVIS (Transitions and Flow VISualization). It provides visualization of data transitions and data flow of Java programs. In the evaluation experiment, we confirmed that by using TFVIS we can reduce the time required to identify defects by about 35%. Hence, it can be said that visualization by TFVIS is effective to improve debugging efficiency.

Keywords: debug, visualizing tool, dynamic analysis, data transitions, Java

1. はじめに

ソフトウェア開発におけるデバッグは，手間のかかる工程である [1]．プログラムの故障により期待した結果を得られない場合，故障の原因である欠陥を特定する必要がある．しかし，この欠陥特定の作業は困難である [2]．

欠陥特定には，デバッグ対象のプログラムが実行時にとる挙動の把握が重要である．プログラマが期待したプログラム実行時の挙動と実際の挙動が乖離する箇所や，異常な値の生成箇所を把握できれば，欠陥を特定するうえで有用な情報になる．また，処理どうしの影響関係を把握することによって，欠陥が存在する疑いがある箇所を絞り込める．

しかし，欠陥を含むプログラムの実行時の挙動把握は困難である [3]．欠陥を含むプログラムを実行した場合，欠陥から始まった異常が後の処理に感染し，異常が拡大する．そのため，欠陥を含むプログラムの実行時の挙動は，プログラマが期待した挙動と大きく乖離する場合がある．プログラム実行時の挙動は一般的に不可視であり，その挙動がどこまで正しかったのかを把握することは困難である．

プログラム実行時の挙動を解析する動的解析には，多くの手法が存在する [4]．特にトレーサでは，プログラム実行

1　宮崎大学
University of Miyazaki, Miyazaki 889–2192, Japan
2　東京工科大学
Tokyo University of Technology, Hachioji, Tokyo 192–0982, Japan
a)　kat@cs.miyazaki-u.ac.jp

© 2018 Information Processing Society of Japan

のすべてのイベントをトレースとして記録することで，大域的なデータと制御の流れを取得することができる [5], [6]．これにより，実行の順序を遡りながら情報を集めることで，逆戻りデバッグのように，効果的に欠陥を特定することが可能である．しかしながら，これらの手法を用いても，プログラム実行時の挙動把握には手間がかかる．

そこで，本研究では，Java プログラムのデバッグの効率化を目的として，プログラムの欠陥特定を支援するデータ遷移可視化ツール TFVIS（Transitions and Flow VISualization）を開発する．TFVIS は，データ遷移可視化 [7], [8] と実行フロー可視化 [9] によって，プログラム実行時の挙動把握を支援する．なお，本研究で対象とする欠陥の種類は，機能性の欠陥，特に，ロジックに関する欠陥である．

データ遷移可視化は，プログラム実行時の挙動を詳細に可視化する．このデータ遷移可視化は，変数更新やループ処理，条件分岐，メソッド呼び出しといったプログラムの主要な処理を組み合わせて図表を作成する．そして，その図表上で，データ遷移を表現することによって，データ遷移をプログラムの他の処理と関連付けて可視化する．このデータ遷移可視化により，データ遷移を効率的に把握し，欠陥特定に活用できるようになる．

実行フロー可視化は，プログラムの実行全体の流れを可視化し，プログラム実行時の挙動把握を支援する．データ遷移可視化は，一度に可視化できる範囲が狭く，可視化箇所の選択が必要である．この実行フロー可視化は，データ遷移可視化を行う箇所の選択を助け，規模の大きいプログラムでのデータ遷移可視化の効果的な活用を可能にする．

TFVIS の可視化により，欠陥を含んだプログラムの実行時の挙動把握を容易にし，欠陥特定を支援する．

2 章では，従来のデバッグ支援手法とその問題点について述べる．また，従来手法の問題点に対し，提案手法が解決する問題について述べる．3 章では，本研究における提案手法について説明する．4 章では，データ遷移可視化ツール TFVIS の機能とデータ遷移の取得について説明する．5 章では，欠陥を含んだ 2 種類のプログラムを TFVIS で可視化し，その効果を確認する．6 章では，TFVIS の評価実験を行う．

2. 従来のデバッグ支援手法

本章では，まず従来のデバッグ支援手法とその問題点について述べる．

プログラムの欠陥特定は，ソースコードの読み直しによって行われる．その際に，プログラム実行時の挙動を把握することによって，欠陥が存在する疑いがある箇所を絞り込むことができ，より効率的に欠陥特定を行える．プログラム実行時の挙動は一般的に不可視であるが，動的解析によって把握できる．動的解析を行うデバッグ支援手法は多く存在する．そういった手法を活用し，プログラム実行時の挙動を把握することによって，欠陥を効率的に特定できる．

これらの支援手法や支援手法を活用したデバッグ支援ツールとその問題点について，以下で述べる

2.1 ブレークポイント

ブレークポイントは，最も多用されるデバッグ支援手法の 1 つである [10]．ブレークポイントは，プログラムの実行を任意の箇所で停止し，停止した時点での変数の値などの状況を確認できるデバッグ支援手法である．

ブレークポイントの問題点は，設置箇所の選定が難しいことである．欠陥の見当がついていない場合，ブレークポイントの設置と再実行の繰り返しによって，プログラム実行時の挙動の情報を集めていかなければならない．ブレークポイントの設置は，デバッグを行うプログラマの能力への依存が大きく，プログラマによっては，関係のない箇所への設置を繰り返してしまう可能性がある [11]．

さらに，ブレークポイントで停止した箇所が，どういう経緯で実行されたのかが分かりにくいという点も問題である．ループ処理や，何度も呼び出されるメソッドの中に，ブレークポイントを設置した場合は，同じコードで何度も停止することになる．この場合，各コードの実行の経緯が分かりにくいため，停止した箇所で実行する処理を判別しにくい．

また，ブレークポイントの再設置や再実行は手間であるだけではない．プログラムによっては，再実行によって処理が変わり，期待した情報を得られなくなる可能性もある．

2.2 プログラムスライシング

プログラムスライシングはプログラムの解析を行う手法の一種である [12]．

プログラムスライシングは，ソースコードの解析を行う際に，調べる必要のある箇所を絞り込む手法である．プログラムスライシングでは，まずユーザが解析の基点としてソースコードの中から変数を選ぶ．そして，その基点の変数の値の生成に影響を与える処理をスライスとして抜き出す．つまり，不審な値の生成原因を調べる場合に，その値の生成に関係したコードだけを抽出できる．

プログラムスライシングには，様々な異なる手法が存在し，手法によって，スライスの抽出対象や，活用方法が異なる．

プログラムスライシングの手法の 1 つに，動的スライシング [13] がある．動的スライシングは，プログラム実行時の挙動について解析する手法であり，データ遷移の解析に活用できる．

しかし，動的スライシングによるデータ遷移解析には，スライスの実行時の状況が分かりにくいという問題がある．同じメソッドが何度も呼び出される場合，あるスライ

© 2018 Information Processing Society of Japan

スがどの呼び出しで実行されたのかが判別しにくい．また，ループ処理のように同じ行を何度も実行する場合は，同じ行が何度もスライスとして抽出される．その結果，各スライスを，どういう状況で実行しているのか判別が難しくなり，スライスどうしの関連性も正しく把握することが難しくなる．

2.3 トレーサ

トレーサは，プログラム実行時のデータの流れや制御の流れを再現し，プログラム実行時の挙動把握を支援するデバッグ支援ツールである [5], [6]．トレーサは，プログラムを最後まで実行し，その実行に関する情報をまとめて記憶することで，メソッドの呼び出し関係や変数の更新履歴の表示を可能にする．そのため，プログラム実行時の特定の処理を基点として，処理を進めながら，もしくは以前の処理に遡りながら，処理や変数更新の流れといったプログラム実行時の挙動を柔軟に解析できる．

櫻井らは，既存のトレーサの問題点として，メソッドをまたぐデータの受け渡しのような，大域的な追跡が困難であることをあげた [6]．この問題を解決するために，櫻井らは，トレースを用いた欠陥の新たな発見手法を提案し，発見手法を実現するためのトレーサである Traceglasses を実装した．

Traceglasses の問題点は，変数どうしの依存関係の把握が困難な点である．不審な変数の値を発見した際に，その原因を探る場合，変数の値の生成に関係した変数を調査する必要がある．トレーサでは，変数の更新履歴を表示することができるが，ある変数の更新に関わった変数をどこで生成したのか確認する機能を持たない．そのため，更新に関わった変数が正しいかを調査するために，検索やステップ実行を繰り返し行う必要がある．

また，注目する処理の前後の詳細な流れの把握が難しいという問題点も存在する．たとえば，ループ処理中に不審な値を発見した際に，ループ処理を何回行ったのか，何回目のループ処理で不審な値を生成したのかという情報は，欠陥を特定するうえで非常に重要な情報となる．トレーサでこのような情報を取得する場合，ループ回数をカウントする変数の値を確認するために，検索やステップ実行を繰り返し行う必要がある．

3. 提案手法

本研究では，Java プログラムのデータ遷移の可視化によって欠陥特定を支援する．

本章では，まず初めに，本研究で提案する手法が対象とするバグの種類について述べる．次に，データ遷移について説明し，データ遷移が欠陥特定にどのように役立つのか詳しく説明する．最後に，提案手法による従来手法の問題点の解決について述べる．

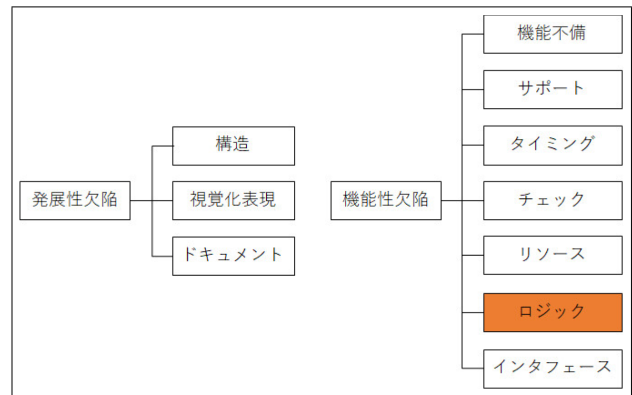

図 1　提案手法が対象とする欠陥の種類
Fig. 1　Scope of defects for our proposed method.

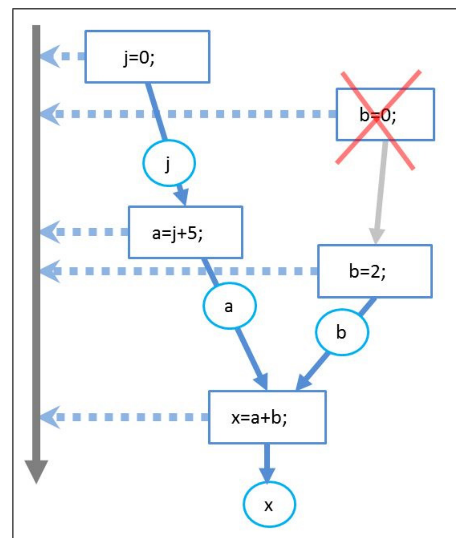

図 2　データ遷移
Fig. 2　Data transitions.

3.1 提案手法が対象とするバグの種類

図 1 に，本研究で提案する手法が対象とする欠陥の種類を示す．これは，論文 [14] の欠陥分類に基づいている．

提案手法が対象とする欠陥の種類は，機能性の欠陥，特にロジックに関する欠陥である．

3.2 データ遷移とは

データ遷移とは，プログラム実行時のデータの流れであり，変数の値どうしの影響関係を意味する．ある変数の値を基点とした場合に，その値の生成に関係した変数の値が存在するならば，その 2 つの値にはデータ遷移の関係がある．図 2 に，データ遷移の例を示す．図 2 は変数 x の値を基点とした際のデータ遷移を図示したものである．基点の変数 x の値は，「x=a+b;」というコードで生成する．このコードで参照した変数 a と変数 b の値は，変数 x の生成に関わっており，基点との間にデータ遷移が存在する．さらに，変数 a の値の生成に，変数 j の値が関わっている．そのため，この変数 a の値と変数 j の値にはデータ遷移が

© 2018 Information Processing Society of Japan

存在し，変数 x の値と変数 j の値の間にも変数 a の値越しにデータ遷移が存在する．

このデータ遷移の把握は，プログラム実行時の挙動把握に効果的である．データ遷移を把握することによって，プログラム実行時に，ある値を生成したのは，どこの処理でいつ起きたことなのか．そして，その値の生成に関わった変数の値は，どこで生成したのかといったプログラム実行時の挙動を把握することが可能になる．

3.3 提案手法による従来手法の問題点の解決

本節では，本研究で提案する手法が解決する従来手法の問題点について述べる．

本提案手法は，2.3 節で述べた既存のトレーサの問題点である，以下の 2 つを解決する．

- 詳細な処理の流れの把握の支援
- 変数の依存関係の把握の支援

本提案手法では，詳細な処理の流れの把握の支援のために，ループ処理の挙動や変数更新といったプログラム実行時の詳細な処理を，図表の形式で可視化する．この可視化により，ユーザはプログラム実行時の処理を視覚的に把握しやすくなる．また，図表化することで，特異な挙動を発見しやすくし，どこの挙動に注視するかを決めやすくする．

次に，プログラム実行時の処理を可視化した図表のうえで，データ遷移を可視化する．このデータ遷移の可視化によって，変数どうしの依存関係を図表上で矢印を用いて明らかにすることで，変数の依存関係の把握の支援を行う．また，変数どうしの依存関係に，異なるメソッドで生成した変数が関わってくる場合，その変数をいつ，どこで生成したのかを示すことによって，不審な値を発見した際の調査に役立つ．

なお，本提案手法において，現状ではプリミティブ型だけが取得可能である．オブジェクトなどが持つデータを可視化するためには，より効果的なデータの収集方法が必要である．

4. TFVIS

本章では，提案手法を実装した，データ遷移可視化ツール TFVIS（Transitions and Flow VISualization）について述べる．TFVIS は，Java プログラムのソースコードを読み込み，実行することで，データ遷移と実行フローを可視化する．**図 3** に，TFVIS の外観を示す．図 3 において，画面右に 2 つ表示している小ウィンドウがデータ遷移図であり，データ遷移図の左隣の図が実行フロー図である．

以下に，TFVIS の各機能について述べる．

4.1 データ遷移可視化

この節では，TFVIS によるデータ遷移可視化について解説する．

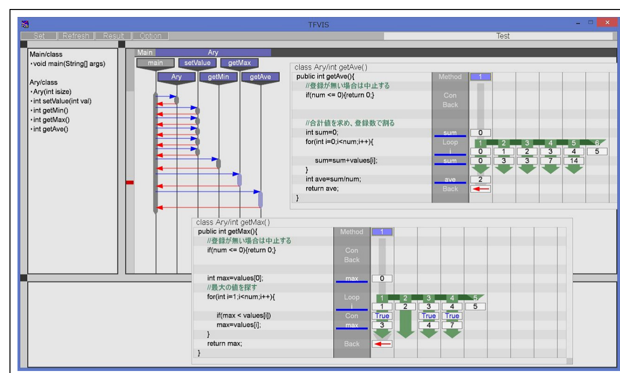

図 3 TFVIS の画面例
Fig. 3 Overview of TFVIS.

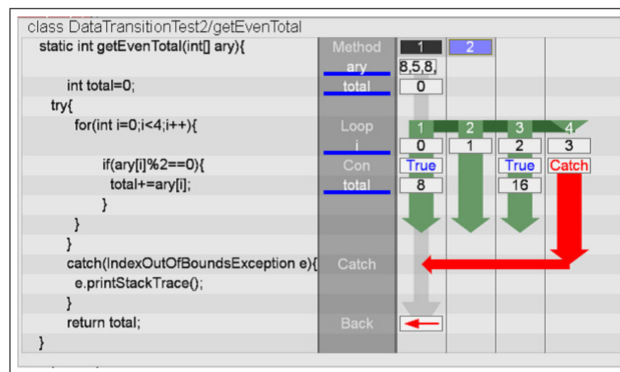

図 4 データ遷移図の例
Fig. 4 Example of data transitions diagram.

4.1.1 データ遷移図

TFVIS は，対象ソースコードを読み込み，静的解析を用いて構造情報を，動的解析を用いて実行時の情報を収集し，データ遷移図を生成する．データ遷移図は，プログラム実行時の挙動を詳細に表す図である．データ遷移図の例を，**図 4** に示す．

データ遷移図は，TFVIS のウィンドウの中に，小ウィンドウとして表示する．また，図 3 に示すように，TFVIS は異なるデータ遷移図を同時に表示できる．

1 つのデータ遷移図が，1 回のメソッドの実行を表している．データ遷移図上では，左から順に，メソッドのソースコード，各行のイベント，更新値と処理の流れを表示する．更新値は，変数更新後のその変数の値を意味し，データ遷移図上に白い背景色の図形の中に，黒文字で表示する．図 4 を例に，データ遷移図の見方について説明する．

図 4 中のソースコードの 2 行目の「int total=0;」の行のように，変数更新を行う行では，実行した際のその変数の更新値を記述したボックスを表示する．データ遷移図中央の背景色が濃い灰色になっている列は，その行にイベントが存在する場合に，そのイベントに関しての情報を表示する．変数の更新を行う場合は，更新の対象になる変数の名前を表示する．そして，変数名の右の列から，更新値を表示する．更新値を表示するのは，それぞれの行で変数の更

© 2018 Information Processing Society of Japan

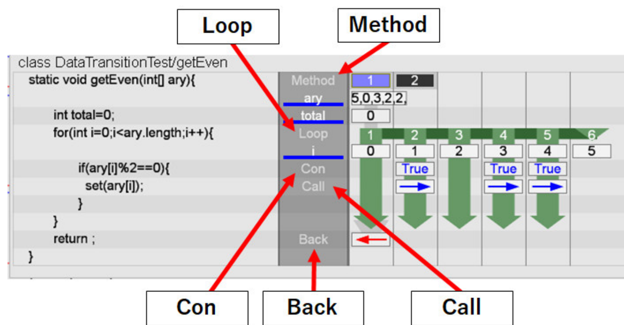

図 5　データ遷移図上でのイベントの表示例

Fig. 5　Example of drawing events on data transitions diagram.

新が起きる場合に限る．2 行目では，変数 total に 0 を代入しているため，更新値 0 を記述したボックスを表示する．

1 行目のメソッドの引数のように，配列を更新する場合は，配列の要素の値をまとめて表示する．1 行目の例のように，メソッドの引数も変数の更新と見なし，更新値を表示する．

ループ処理により，同じ行を何度も実行する場合は，更新値を右の列にずらして表示する．4 行目では，変数 i の更新を行っている．しかし，for 文によるループ処理により，変数 i の更新は何度も起きる．このような場合は，更新値を右にずらして配置していく．

また，ループの同じ周回での更新値は，同じ列に配置する．6 行目の変数 total の更新は，if 文の中にある処理である．そのため，for 文の繰り返しの中で実行する場合と，実行しない場合がある．この例では，変数 total の更新は for 文の 1 週目と 3 週目に実行する．更新値とループとの関係を分かりやすくするために，ループの同じ周回での更新値は，同じ列に配置する．

緑色の下向きの矢印が，ループの挙動を表す．矢印の中に白色で表示した数字は，ループの周回番号を表す．矢印で強調している範囲が，ループの各周回の処理を表す．

このように，データ遷移図上にループの挙動を図示することによって，更新値とループ処理の関係が一目で把握できる．

例外処理によって，処理の流れが変わる場合，例外処理発生の箇所に赤で「Catch」と記述したボックスを表示し，そこから例外処理実行の箇所まで赤い矢印を表示する．図 4 では，5 行目の if 文の条件式「ary[i]%2==0」において，要素数 3 の配列の 4 番目を参照していることから例外が発生している．この例のように，ループ処理中など，例外処理実行の列と例外処理実行の列が異なる場合，縦方向と横方向の 2 つの矢印で表示を行う．

ウィンドウ中央の濃い灰色の列には，変数更新といったイベントの情報を表示する．各イベントは，**図 5** に示す例のように，データ遷移図上に表示する．データ遷移図上に表示するイベントを，**表 1** に示す．

表 1　データ遷移図上のイベント

Table 1　Events on data transitions diagram.

イベント名	データ遷移図表記
メソッド呼び出し	Call
メソッド開始	Method
メソッド終了	Back
ループ開始	Loop
条件成立	Con

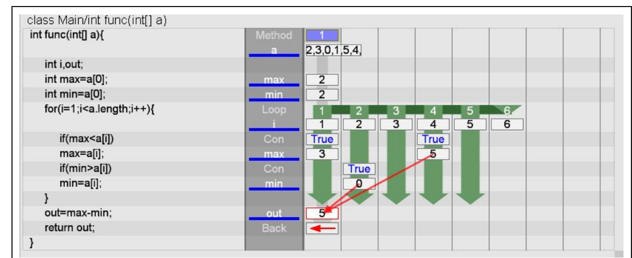

図 6　データ遷移線の表示例

Fig. 6　Example of drawing data transitions lines.

メソッド呼び出しの行では，データ遷移図上に，メソッド呼び出しを表す青い矢印を表示する．

メソッド開始の行の「Method」の右の数字は，メソッド別の呼び出し番号を表している．図 5 のデータ遷移図では，Method 表記の右に 1 と 2 を表示している．これは，図 5 のデータ遷移図が可視化しているメソッドを，プログラムの実行全体で 2 回呼び出していることを表す．呼び出し番号の背景色は通常は黒であるが，データ遷移図が可視化している箇所に対応する呼び出し番号は，青い色の背景色で表示する．

メソッド終了の行では，データ遷移図上に，処理がメソッドの呼び出し元に戻ることを表す赤い矢印を表示する．

ループ開始の行では，ループの周回の回数を表示する．

条件成立の行で条件式が成立した場合，データ遷移図上に「True」と表示し条件式の成立を表す．

4.1.2　データ遷移線

データ遷移線は，データ遷移を可視化する機能である．ユーザが，データ遷移図上で変数の不審な値を発見した場合に，データ遷移線を活用することによって，不審な値を生成した原因の特定が容易になる．

図 6 に，データ遷移線の表示例を示す．図 6 を用いて，データ遷移線の見方について解説する．図 6 のデータ遷移図上を横切る 2 本の赤い矢印がデータ遷移線である．データ遷移線の基点の更新値は，赤い枠線で表示する．図 6 の例では，変数 out の更新値がデータ遷移線の基点である．基点とデータ遷移線でつながっている更新値は，変数 max の更新値と，変数 min の更新値である．この 2 つの更新値が，基点での処理「out=max−min;」で参照した変数 max と変数 min の更新値であることを示している．

データ遷移線の表示には，基点の選択が必要である．ユー

© 2018 Information Processing Society of Japan

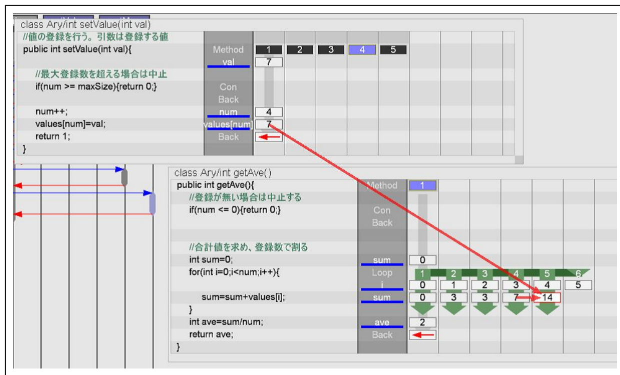

図 7　メソッド間のデータ遷移を可視化するデータ遷移線の表示例
Fig. 7　Example of drawing data transitions line between methods.

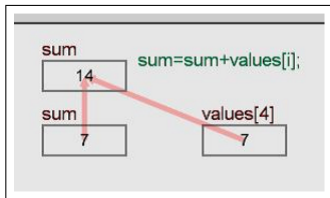

図 8　データ遷移の注釈の表示例
Fig. 8　Example of drawing annotation of data transitions.

ザは，データ遷移図上の更新値からデータ遷移線の基点を選択する．配列のように，複数の更新値を 1 つのセルに表示している場合でも，配列の各要素を個別に選択できる．

データ遷移線は，異なるメソッド間でのデータ遷移を可視化できる．**図 7** に，メソッド間のデータ遷移を可視化するデータ遷移線の表示例を示す．図 7 の例では，データ遷移線の基点に変数 sum の更新値を選択している．この変数 sum の値は，変数 sum と変数 values[i] の値を用いて生成する．変数 values[i] の値を生成するメソッドは，データ遷移線の基点が所属するメソッドと異なるが，正しくデータ遷移線を表示できている．

データ遷移線の利便性を高めるために，データ遷移の注釈を表示する機能を実装する．データ遷移の注釈は，TFVIS のメインウィンドウの下部に表示する．**図 8** に，データ遷移の注釈の表示例を示す．図 8 の例では，図 7 の例と同様に，データ遷移線の基点に変数 sum の更新値を選択している．図 8 の注釈は，変数 sum の値 7 と，変数 values[4] の値 7 を参照して，変数 sum の値を 14 に更新したことを示している．「values[i]」のような変数の指定は，プログラムの実行状況で指し示す対象が変わる．このような変数は，データ遷移図やデータ遷移線では，具体的にどの変数なのかが把握しにくい．そのため，データ遷移の注釈には，図 8 に示すように，実際に使用した変数の名前を表示する．このデータ遷移の注釈の表示機能を活用することによって，よりプログラム実行時の挙動把握が容易になる．

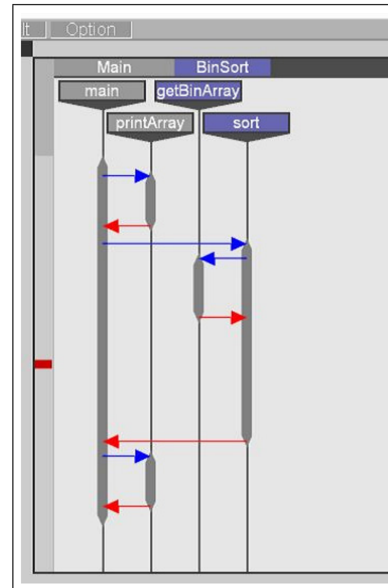

図 9　実行フロー図
Fig. 9　Executional flow diagram.

4.1.3　データ遷移可視化の問題点

データ遷移可視化は，一度に可視化できる範囲が狭いという問題がある．データ遷移可視化は，ソースコードの各行ごとに情報を表現する．しかし，このような可視化は大量の処理の可視化には適していない [15]．そのため，データ遷移可視化は，プログラム実行時の処理全体を一度には可視化できない．

データ遷移可視化の活用には，可視化を行う箇所の選択が必要である．上述した問題により，本研究でのデータ遷移可視化はメソッドの各呼び出しごとに可視化する．そのため，ユーザはデータ遷移図として表示する箇所を選ぶ必要がある．しかし，可視化対象のプログラムが多数のメソッドを持っている場合は，可視化箇所の適切な選択は難しい．

そこで，これらの問題点を解決するため，実行フローの可視化を行う．

4.2　実行フロー可視化

この節では，実行フロー可視化について説明する．実行フロー可視化は，プログラムの実行全体の流れを可視化し，実行フロー図を生成する．ユーザは，実行フロー図を用いることによって，データ遷移可視化をより効果的に活用できる．

4.2.1　実行フロー図

図 9 に，実行フロー図の例を示す．実行フロー図は，可視化対象のプログラムの実行全体の流れを示す．実行フロー図は，UML（Unified Modeling Language）のシーケンス図 [16] を基にしており，各クラスのメソッドの使用状況や，メソッド呼び出しの関係を表す．ソースコードの各行ごとに情報を示すデータ遷移図に対し，実行フロー図は

© 2018 Information Processing Society of Japan

メソッド単位で情報を示す．そのため，変数の値の遷移といった細かい情報を表現できないが，プログラム実行時の処理全体を一度に可視化できる．

図 9 を例に，実行フロー図の見方を説明する．実行フロー図の最上段には各クラスの名前を表示している．その下には，各メソッド名を表示している．各メソッドから出ている黒い線はメソッドのライフラインであり，ライフライン上の太い部分はメソッドの活性区間を表す．初期設定では活性区間を灰色で表示するが，データ遷移図を表示している箇所に対応する活性区間は薄い青色で表示する．青い矢印は，メソッドの呼び出しを表す．赤い矢印は，メソッドの処理の終了を表す．

図 9 でのメソッド「sort」の挙動を例にすると，以下の情報を読み取れる．メソッド「sort」は，メソッド「main」が，メソッド「printArray」の呼び出しの後に呼び出している．また，メソッド「sort」は自身の処理の中で，メソッド「getBinArray」を呼び出している．

実行フロー図は，どのメソッドがどういうメソッドを呼び出すかといった挙動の把握に活用できるが，呼び出しがないメソッドや，異常な回数の呼び出しを受けるメソッドといった異常の判別にも活用できる．

実行フロー図とデータ遷移図を併用することによって，より効果的にプログラム実行時の挙動を把握できる．実行フロー図とデータ遷移図を併用することによる利点は，主に 2 つある．以下に，各利点について述べる．

第一に，データ遷移図の可視化箇所の選択が容易になる．実行フロー図を用いることによって，プログラムの実行の流れを把握できる．各メソッドを，どのような流れで呼び出しているのかを把握できるため，データ遷移図の可視化箇所を選びやすい．また，実行フロー図上の活性区間から，対応するデータ遷移図を表示できるため，操作の手間も少ない．

第二に，データ遷移をより把握しやすくなる．データ遷移線は，メソッド間のデータ遷移を可視化できる．実行フロー図を用いることによって，互いが属するメソッド呼び出しの関係を把握しやすくなる．

データ遷移線の表示に，実行フロー図の活性区間を用いる場合がある．図 7 のデータ遷移線は，メソッド間のデータ遷移を可視化している．このようにデータ遷移線を表示できる理由は，データ遷移線の起点側のデータ遷移図と，終点側のデータ遷移図の両方を表示しているからである．そのため，起点側か終点側のどちらかのデータ遷移図を表示していない場合は，対応する更新値の座標を取得できないため，更新値どうしを結ぶデータ遷移線を表示できない．このような場合の対処として，データ遷移図の更新値ではなく実行フロー図の活性区間をデータ遷移線の基点に用いる．このようにすることによって，対応するデータ遷移図を表示していない場合でも，メソッド間のデータ遷移を可視化できる．データ遷移図の更新値と実行フロー図の活性区間をつなぐデータ遷移線の例を，**図 10** に示す．

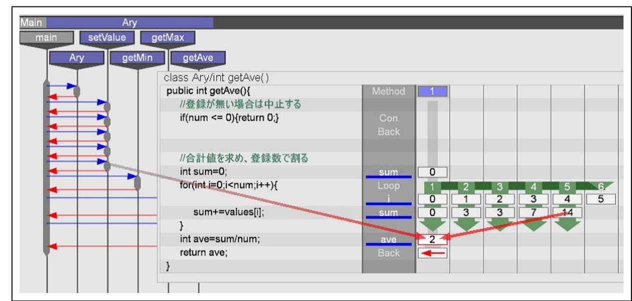

図 10　更新値と活性区間をつなぐデータ遷移線の例

Fig. 10 Example of data transitions lines connecting update values and method active section.

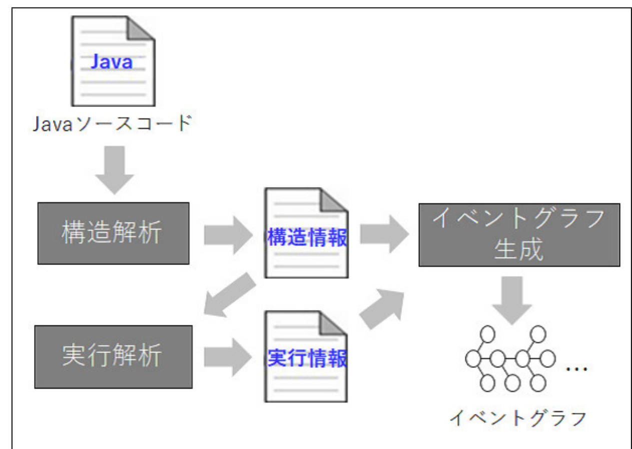

図 11　イベントグラフ生成の流れ

Fig. 11 Flow of generating event graph.

図 10 での，データ遷移線の基点は，変数 ave の更新値である．変数 ave の値の生成には，変数 sum と変数 num の値を用いるが，変数 num は，別のメソッドで値を更新している．また，変数 num の値を更新したメソッド「setValue」に対応するデータ遷移図を表示していない．図 10 では，変数 num の更新値の代わりに，メソッド「setValue」の活性区間からデータ遷移線を表示しており，対応するデータ遷移図を表示していない場合でも，メソッド間のデータ遷移を可視化できている．

4.3 データ遷移の取得

本節では，データ遷移の取得方法について説明する．データ遷移は，実行したイベントの関係を表すイベントグラフの生成によって取得する．**図 11** に，イベントグラフ生成の流れを示す．

以下で，各処理の詳細について述べる．

4.3.1 構造解析

構造解析では，プログラムの構造の解析を行い，構造情報を取得する．構造情報として，Java ソースコードの各行ごとに以下の情報を取得する．

© 2018 Information Processing Society of Japan

- イベント種別
- イベントの対象
- イベントの関係者
- 表示コード

イベント種別は，変数更新や変数参照，メソッド呼び出し，分岐処理，ループ処理開始など，可視化の基準となる特定の処理である．イベントの対象は，イベントで影響を受ける対象を指す．イベントの関係者は，イベントに影響を与える対象を指す．表示コードは，ツール上で表示するソースコードを指す．

例として，“a=b+c;” というコードについて，どのような情報を抽出するか説明する．発生するイベントは，変数更新である．イベントの対象は，更新を受ける変数 a である．イベントの関係者は，更新時に参照する変数 b と c である．表示コードは，“a=b+c;” である．

なお，構造解析の際，オブジェクトの生成については，変数の更新と見なしていない．これは，プリミティブ型の変数と同じようにオブジェクトの値を得ようとすると，オブジェクトを表す意味のない数値を得てしまうためである．したがって，オブジェクトからある値を変数に代入した際には，オブジェクトが含む変数の値を取得することができるが，ある時点のオブジェクトの内部状態を一覧化する，といったことが現時点では不可能である．

4.3.2 実行解析

実行解析では，可視化対象のプログラムの実行時の挙動を解析し，実行情報を取得する．実行情報の取得は，ソースファイルに対するプローブの埋め込みによって取得する．

構造解析によって得た，各行のイベントごとに対応したプローブの挿入を行う．イベントに対応したプローブは，そのイベントを可視化する際に必要となる情報を出力する．たとえば，変数更新イベントに対応して埋め込む変数更新検出プローブは，実行時のイベント ID やクラスインスタンス番号，メソッド識別番号，メソッド実行番号，行番号に加えて，更新の対象になった変数，更新後の変数の値，更新に影響を与えた変数の情報を出力する．

4.3.3 イベントグラフの生成

構造情報と実行情報を基に，イベントグラフを生成する．イベントグラフは，実行情報を基に，各イベントを実行順に，グラフのノードとして生成する．この際に，各ノードはイベント固有の変数やオブジェクトを保持する．

たとえば，変数更新イベントのノードを生成した際には，ソースコード上の静的な位置，変数の値，前の変数更新イベントのノード，次の変数更新イベントのノード，更新に影響を与えた変数の変数更新イベントのノードの位置を持つ．更新に影響を与えた変数の変数更新イベントのノード位置は，実行情報で得た更新に影響を与えた変数の情報から，同一クラス内を検索することで得る．

このようなノード間のつながりによって，データ遷移を表現する．

5. 適用例

本章では，実際に TFVIS でプログラムの欠陥特定を行い，TFVIS による可視化の効果を確認する．適用例には，以下に示す 2 種類のプログラムを用意した．

(1) 配列計算プログラム
(2) 在庫管理プログラム

これらのプログラムは，それぞれが異なる欠陥を含んでおり，それらの欠陥特定を TFVIS を活用して行う．配列計算プログラムの適用では，変数の依存関係の把握の支援ができることを示す．在庫管理プログラムの適用では，詳細な処理の流れの把握の支援ができることを示す．

5.1 配列計算プログラム

図 12 に，この配列計算プログラムのソースコードを示す．このプログラムに値を登録すると，クラス「Ary」のメンバである int 型配列 values の先頭の要素から，その値を代入していく．そして，登録した値の集合の最小値，最大値，平均値を計算する．図 12 のメソッド「setValue」が，値を登録するメソッドであり，メソッド「getMin」が最小値，メソッド「getMax」が最大値，メソッド「getAve」が平均値を計算するメソッドである．なお，このプログラムでは，平均値を整数で出力する．

```
1  public class Ary {
2
3    int num;
4    int maxSize;
5    int[] values;
6    //メンバの初期化を行う． 引数は最大登録数
7    public Ary(int isize){
8      num=0;
9      maxSize=isize;
10     values = new int[maxSize];
11   }
12   //値の登録を行う． 引数は登録する値
13   public int setValue(int val){
14
15     //最大登録数を超える場合は中止する．
16     if(num >= maxSize){return 0;}
17     num++;
18     values[num]=val;
19     return 1;
20   }
21   public int getMin(){
22     //省略
23   }
24   public int getMax(){
25     //省略
26   }
27   public int getAve(){
28   //登録が無い場合は中止する
29     if(num <= 0){return 0;}
30     //合計値を求め，登録数で割る
31     int sum=0;
32     for(int i=0;i<num;i++){
33       sum=sum+values[i];
34     }
35     int ave=sum/num;
36     return ave;
37   }
38
39 }
```

図 12 配列計算プログラムのソースコード
Fig. 12 Source code of array calculation program.

© 2018 Information Processing Society of Japan

情報処理学会論文誌　Vol.59　No.4　1137–1149 (Apr. 2018)

```
1  public class Main {
2
3    public static void main(String[] args) {
4
5      Ary ary=new Ary(7);
6
7      //値の登録
8      ary.setValue(3);
9      ary.setValue(0);
10     ary.setValue(4);
11     ary.setValue(7);
12     ary.setValue(2);
13
14     int min=ary.getMin();
15     System.out.println(‘‘最小値>>’’+min);
16
17     int max=ary.getMax();
18     System.out.println(‘‘最大値>>’’+max);
19
20     int ave=ary.getAve();
21     System.out.println(‘‘平均値>>’’+ave);
22   }
23 }
```

図 13　配列計算プログラムのテスト用コード
Fig. 13　Test code of array calculation program.

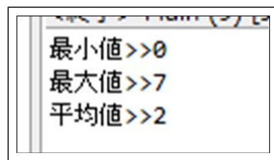

図 14　配列計算プログラムのテスト用コードの実行結果
Fig. 14　Execution result of the test code of array calculation program.

この配列計算プログラムが含む欠陥は，「setValue」での配列への値の代入時に，配列番号の指定を間違えていることである．図 12 の 18 行目で，変数 num を配列番号の指定に用いて，配列要素の先頭から値の代入を行っている．この変数「num」は，現在の値の登録数であり，「setValue」を呼び出すたびに値に 1 を加算する．この加算を，配列への値の代入の前に行っているので，配列への値の代入が，配列要素の 0 番目からではなく，配列要素の 1 番目からの代入になっている．それに対し，「getMin」，「getMax」，「getAve」といったメソッドでは配列要素の 0 番目から値を参照しているため，期待出力と異なる出力結果になる．

この配列計算プログラムを，テスト用コードを用いて実行する．**図 13** に，配列計算プログラムのテスト用コードを示す．このテスト用コードでは，3，0，4，7，2 の値を登録し，最小値，最大値，平均値を出力する．

テスト用コードの実行結果を，**図 14** に示す．最小値と最大値は正しいように見えるが，平均値の期待値は 3 であり，このプログラムが故障していると判断できる．

実行結果では，平均値のみが期待値と異なる．そのため，メソッド「getAve」が欠陥を含んでいると予想し，「getAve」のデータ遷移図を確認する．「getAve」のデータ遷移図を，**図 15** に示す．テスト用コードで，登録した値を合計すると 16 である．図 15 のデータ遷移図を確認すると，値の合計値が入る変数 sum の最終的な値が 16 ではないことが

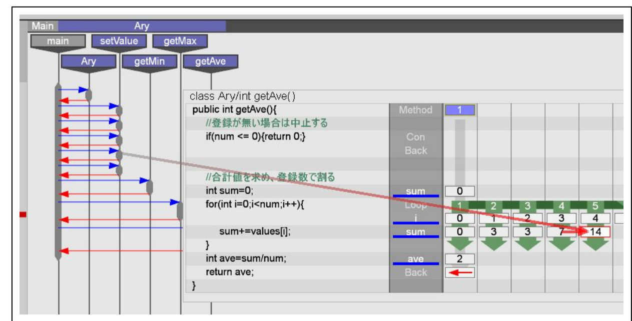

図 15　メソッド「getAve」のデータ遷移図
Fig. 15　Data transitions diagram of method ‘getAve’.

分かる．変数 sum の値の遷移を確認すると，for 文の 4 周目での変数 sum の値から，5 周目での変数 sum の値は，7 だけ増えていることが分かる．しかし，テスト用コードでの最後の値の登録は，2 であり，処理に食い違いが生じている．そこで，for 文の 5 周目での変数 sum の値を基点にデータ遷移線を表示する．表示したデータ遷移線を確認すると 2 本のデータ遷移線が存在し，片方は 4 周目の変数 sum の値，もう片方は「setValue」の 4 度目の実行を指していることが分かる．ここで，配列への値の登録が 1 つずれていることが分かり，「setValue」での配列への値の代入に問題があると分かる．

5.2　在庫管理プログラム

在庫管理プログラムは，在庫の追加，在庫数の操作，在庫の検索，単価計算の機能を持つ．テストシナリオとして，新しい在庫を 4 つ追加し，在庫数を操作したうえで，単価計算を行う操作を仮定した．以下に，テストシナリオの詳細な手順を示す．

(1) 在庫「apple」を 50 kg，総額 7,500 円で登録
(2) 在庫「banana」を 40 kg，総額 8,000 円で登録
(3) 在庫「cherry」を 30 kg，総額 90,000 円で登録
(4) 在庫「durian」を 10 kg，総額 10,000 円で登録
(5) 在庫「cherry」の在庫数を 30 kg 減らす
(6) 各在庫の単価を計算

単価の計算を行った結果，正しく各在庫の単価が算出されなかった．単価計算行うメソッド「calculateUnitPrice」のソースコードを**図 16** に，在庫管理プログラムの出力結果を**図 17** に示す．

単価の計算では，在庫が存在しない場合 0 除算が発生する可能性があるため，例外処理を行った．図 16 に示すように，「calculateUnitPrice」メソッドでは，0 除算が発生した場合，「在庫なし」と出力する．

図 17 に示すように，「cherry」の在庫数操作を行った後，それぞれの単価を出力しているが，「durian」の単価が出力できていないことが分かる．このことから，メソッド「calculateUnitPrice」が欠陥を含んでいると予想できる．

メソッド「calculateUnitPrice」の可視化を，**図 18** に示

© 2018 Information Processing Society of Japan

情報処理学会論文誌　Vol.59　No.4　1137–1149 (Apr. 2018)

す．このメソッドでは，在庫の回数だけ単価の計算を繰り返す．図 18 から，本来 4 回繰り返すはずの単価の計算の処理が，3 回目の周回で例外処理によってループから抜け出していることが分かる．このことから，try-catch 文の挿入箇所に問題があることが分かる．

以上，TFVIS を活用して，2 種類のプログラムが含む欠陥をすべて特定した．TFVIS が，変数の依存関係の把握の支援ができることを示した．さらに，詳細な処理の流れの把握の支援ができることを示した．

```
1  public void calculateUnitPrice() {
2    Item item;
3    int weight, totalPrice=0,  unitPrice;
4    String name;
5
6    try {
7      for (int i = 0; i < ItemList.size(); i++) {
8        item = ItemList.get(i);
9        name = item.getName();
10       weight = item.getWeight();
11       totalPrice = item.getTotalPrice();
12       unitPrice = totalPrice / weight;
13       System.out.println(name + ‘‘:’’ +
             unitPrice);
14     }
15   } catch (ArithmeticException e) {
16     System.out.println(‘‘在庫なし");
17   }
18     return;
19 }
```

図 16　単価計算メソッド「calculateUnitPrice」のソースコード
Fig. 16　Source code of method ‘calculateUnitPrice’.

```
在庫管理システム
在庫appleを登録しました
在庫bananaを登録しました
在庫cherryを登録しました
在庫durianを登録しました
cherryを-30kg追加しました。
apple:150
banana:200
在庫なし
在庫管理システム終了
```

図 17　在庫管理プログラムの出力結果
Fig. 17　Output of stock control program.

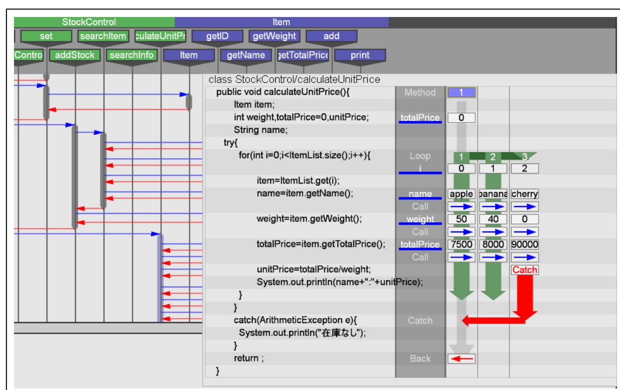

図 18　メソッド「calculateUnitPrice」のデータ遷移図
Fig. 18　Data transitions diagram of method ‘calculateUnitPrice’.

6.　評価

この章では，TFVIS による Java プログラム実行時の挙動可視化が，欠陥特定に有効であることを評価する．初めに，TFVIS の速度性能評価実験について述べる．次に，被験者間実験の評価方法について述べる．そして，評価実験の結果と考察について述べる．

6.1　速度性能評価実験

表 2 に，TFVIS の実行情報取得に関する性能を，**表 3** に，可視化に関する性能をそれぞれ示す．性能速度の計測には，バブルソート（35 行のコード）を繰り返し実行するプログラムを用いた．

また，速度性能評価実験に用いた PC は，CPU が Intel(R) Core(TM) i7-4770，メモリ 8 GB である．

実行情報の取得に関する性能では，バブルソートの呼び出し回数が増えるのに比例して，実行情報を取得する時間が増加していることが分かる．

実行情報の可視化に関する性能では，プログラムの大きさが変わっても，あまり可視化にかかる時間が大きく変わることはなかった．

これらの結果から，実行情報を取得するまでの解析にかかる時間が現実的な数値であれば，TFVIS は適用可能である．したがって，メソッド呼び出し 100 回，もしくは 3,500 行程度以内のプログラムであれば，TFVIS は適用可能であるといえる．

6.2　評価実験方法

TFVIS による欠陥特定の支援効果を確かめるために，被験者を集めて評価実験を行う．この評価実験では，被験者が 2 種のプログラムの欠陥特定を行い，欠陥特定に要した時間を計測する．1 つ目の実験で，被験者らの半数は TFVIS

表 2　実行情報取得に関する性能
Table 2　Time to get execution information.

バブルソート回数	速度
10	1.1 秒
100	4.2 秒
1,000	65 秒

表 3　実行情報の可視化に関する性能
Table 3　Time to visualize execution information.

バブルソート回数	速度
10	1.9 秒
100	2.2 秒
1,000	2.5 秒

© 2018 Information Processing Society of Japan

表 4　1 つ目の評価実験の計測結果
Table 4　Measure time of 1st experiment.

	TFVIS 使用
被験者 A	282 秒
被験者 B	328 秒
被験者 C	297 秒
平均	302 秒

	TFVIS 未使用
被験者 D	478 秒
被験者 E	371 秒
被験者 F	506 秒
平均	452 秒

表 5　2 つ目の評価実験の計測結果
Table 5　Measurement time of 2nd experiment.

	TFVIS 使用
被験者 D	373 秒
被験者 E	854 秒
被験者 F	1,200 秒
平均	809 秒

	TFVIS 未使用
被験者 A	132 秒
被験者 B	1,002 秒
被験者 C	820 秒
平均	651 秒

を使って欠陥特定を行い，残りの被験者らは TFVIS を使わず欠陥特定を行う．その後，TFVIS の有無を交換した 2 つ目の実験を行い，TFVIS の使用の有無で，欠陥特定に要した時間がどのように変わるかを調べる．

この評価実験には，6 人の被験者が参加する．被験者は全員，情報系学科に所属している学生であり，プログラミング言語 Java の利用経験がある．

1 つ目の評価実験では，5.1 節で適用例に用いた配列計算プログラムを用いる．2 つ目の評価実験では，5.2 節で適用例に用いた在庫管理プログラムを拡張したものを用いる．被験者らには，このプログラムの役割や構造について事前に説明を行う．

TFVIS を使用する被験者には，事前に TFVIS の使用方法の説明を行う．具体的には，今回の評価実験で用いるプログラムとは関係性がない，例題用プログラムを用いて TFVIS の機能について説明を行う．

TFVIS を使用しない被験者は，統合開発環境 Eclipse [17] 上で欠陥特定を行う．このとき，手動でのプローブ挿入，Eclipse のデバッグ機能の利用などの欠陥特定を効率的に行うための工夫は自由に行ってよいとする．

計測は，被験者が欠陥を具体的に報告し，その報告が適切であった場合に終了する．また，被験者が欠陥を報告している間は，計測を中断する．欠陥の報告が不適切であった場合は，不適切であると被験者に伝え，計測を再開する．TFVIS の使用の有無にかかわらず制限時間を 20 分とし，計測時間が制限時間を超過した場合は，その時点で計測を終了する．

6.3　評価実験の結果と考察

評価実験の計測結果を，**表 4** と**表 5** に示す．

1 つ目の評価実験では，TFVIS を使用した被験者は，TFVIS を使用しなかった被験者よりも，平均して約 35%欠陥の特定にかかる所要時間が短かった．一方で，2 つ目の評価実験では，TFVIS を使用した被験者は，TFVIS を使用しなかった被験者よりも，平均して約 24%欠陥の特定にかかる所要時間が多かった．表 4 と表 5 の計測結果を，評価実験中の被験者らの行動と，評価実験後の聞きとりから考察する．

1 つ目の評価実験では，TFVIS を使用した被験者は，TFVIS を使用しなかった被験者よりも少ない手間で，プログラム実行時の挙動の情報を得ている．TFVIS を使用しなかった被験者らは，欠陥特定までに，ブレークポイントの設置，プログラムの再実行を繰り返した．それに対し，TFVIS を使用した被験者らは，そういった作業に時間を使わなくても，プログラム実行時の挙動の情報を得ている．例をあげると，TFVIS の有無にかかわらずすべての被験者らは，実験開始直後からメソッド「getAve」のローカル変数である変数 sum の値の遷移に着目した．TFVIS を使用しない場合，プローブやブレークポイントを使った確認が必要である．TFVIS を使用する場合，一目で「getAve」内ループ処理の挙動や変数 sum の値の遷移を，より早く把握することができため，欠陥特定にかかる時間を短縮することができた．

このことから，TFVIS によるプログラム実行時の可視化が，ロジックに関する欠陥の特定までにかかる時間の短縮に効果的であると確認できた．

2 つ目の評価実験では，在庫管理プログラムに，新たに，指定数量以下の在庫すべてに対し，任意の数量を追加する addStockUnderWeight メソッドなど，数値計算を行うメソッドを複数実装した．このプログラムに対し，Item クラスの在庫数を加える addWeight メソッドにバグを埋め込んだ．

この評価実験では，addWeight メソッドに埋め込んだ欠陥が，他の数値計算クラスの存在によって気づかれにくくなっているため，TFVIS による不審な値の追跡が有効であると考えた．

しかしながら，現時点の TFVIS では，オブジェクトの持つデータに対して，データ遷移線の機能を用いることができない．そのため，多くのオブジェクトによって，データ遷移線が遮られ，機能しなくなる結果となってしまった．データ遷移線は，不審な値を追跡する際に非常に強力な機能である．データ遷移線が機能しなくなることで，不審な値を追跡する能力が弱まってしまった．このことが，TFVIS を用いた被験者が，TFVIS を用いなかった被験者に比べ欠陥の発見が遅れてしまった理由だと考える．

オブジェクトの可視化に対応し，データ遷移線の機能をオブジェクトにも対応することで，オブジェクトを多く含むプログラムにおいても，不審な値を追跡する能力を失わずに済むことができる．

これらのことから，オブジェクトの可視化が，TFVIS による欠陥特定の支援に必要であることが分かった．

© 2018 Information Processing Society of Japan

7. おわりに

本研究では，Java プログラムのデバッグの効率化を目的として，プログラムの欠陥特定を支援するデータ可視化ツール TFVIS を開発した．TFVIS は，プログラム実行時の挙動を，データ遷移可視化と実行フロー可視化によってユーザに示す．TFVIS の可視化により，欠陥を含んだプログラムの実行時の挙動把握が容易になり，欠陥を効率的に特定できるようになる．

データ遷移可視化は，プログラム実行時の挙動を詳細に可視化する．変数更新やループ処理，条件分岐，メソッド呼び出しといったプログラムの主要な処理を，組み合わせて図表にすることによって，プログラム実行時の挙動を視覚的に把握できるようにした．

実行フロー可視化は，プログラムの実行全体の流れを可視化する．実行フロー可視化は，メソッド単位でのプログラム実行時の挙動を可視化し，プログラム実行時の挙動把握を支援する．実行フロー可視化を活用することによって，データ遷移可視化を行う箇所を選択しやすくなる．

本研究では，プログラムの欠陥特定を，TFVIS を用いて行い，TFVIS が欠陥特定に効果があることを確認した．そして，実際に被験者を用いて，TFVIS の使用の有無が，欠陥特定に要する時間にどう影響するかを評価実験で計測した．今回の実験結果により，TFVIS を活用することで，機能性，特にロジックに関する欠陥の特定に要する時間を削減できることを確認した．

以上のことから，本研究で開発した可視化ツール TFVIS は，デバッグ工程における欠陥特定の作業を支援することができる．

以下に今後の課題をあげる．

- オブジェクトの可視化
  本論文の TFVIS は，オブジェクトが持つデータを可視化することができない．TFVIS でオブジェクトが持つデータを確認するためには，オブジェクトのデータを変数へ代入する処理が必要となる．プログラムの構造を熟知していない場合，オブジェクト内のデータがどこで更新されたのか把握することは非常に困難である．そのため，オブジェクトとオブジェクト内のデータ遷移の可視化に対応する必要がある．
- マルチスレッドプログラムへの対応
  本論文の TFVIS は，マルチスレッドプログラムを可視化できない．近年，マルチスレッドを用いるプログラムは少なくない．そこで，TFVIS の有用性向上のために，マルチスレッドプログラムへの対応が必要であると考えられる．しかし，本論文の実行フロー図の表現は，マルチスレッドプログラムに対応していない．そのため，実行フロー図の表現を改良する必要がある．

参考文献

[1] LaToza, T.D., Venolia, G. and DeLine, R.: Maintaining Mental Models: A Study of Developer Work Habits, *Proc. 28th International Conference on Software Engineering, ICSE '06*, pp.492–501, ACM (2006).

[2] Pressman, R.S.: *Software Engineering A Practitioner's Approach*, McGraw-Hill Science (2001).

[3] Zeller, A.：デバッグの理論と実践—なぜプログラムはうまく動かないのか，オライリー・ジャパン (2012).

[4] 石尾　隆：プログラムの動的解析，コンピュータソフトウェア，Vol.16, No.5, pp.78–83 (1999).

[5] 柏村俊太郎，寺田　実：実行トレースの特性を生かしたプログラミング支援，情報処理学会夏のプログラミング・シンポジウム「夢をかけるプログラミング—世代を超えて・夢の再発見」報告集，Vol.3, No.3, pp.79–84 (2006).

[6] 櫻井孝平，増原英彦，古宮誠一：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），Vol.3, No.3, pp.1–17 (2010).

[7] 中村紘人，片山徹郎，喜多義弘，山場久昭，岡崎直宣：Java プログラム実行時のデータ遷移可視化によるデバッグ支援，ソフトウェアエンジニアリングシンポジウム 2014 論文集，pp.125–130 (2014).

[8] 佐藤拓弥，片山徹郎，水久保直哉，田中伸英：例外処理を含む Java プログラムを対象としたデータ遷移可視化ツール TFVIS の適用範囲の拡大，ソフトウェア・シンポジウム 2017in 宮崎論文集，pp.28–35 (2017).

[9] Nakamura, H., Katayama, T., Kita, Y., Yamaba, H., Aburada, K. and Okazaki, N.: TFVIS: A Supporting Debugging Tool for Java Programs by Visualizing Data Transitions and Execution Flows, *The 2015 International Conference on Artificial Life and Robotics*, pp.376–379 (2015).

[10] Murphy, G.C., Kersten, M. and Findlater, L.: How Are Java Software Developers Using the Eclipse IDE?, *IEEE Softw.*, Vol.23, No.4, pp.76–83 (online), DOI: 10.1109/MS.2006.105 (2006).

[11] Zhang, C., Yang, J., Yan, D., Yang, S., and Chen, Y.: Automated Breakpoint Generation for Debugging, *Journal of Software*, pp.603–616 (2013).

[12] Weiser, M.: Programmers Use Slices when Debugging, *Comm. ACM*, Vol.25, No.7, pp.446–452 (online), DOI: 10.1145/358557.358577 (1982).

[13] Agrawal, H. and Horgan, J.R.: Dynamic Program Slicing, *SIGPLAN Not.*, Vol.25, No.6, pp.246–256 (online), DOI: 10.1145/93548.93576 (1990).

[14] Mantyla, M.V. and Lassenius, C.: What Types of Defects Are Really Discovered in Code Reviews?, *IEEE Trans. Softw. Eng.*, Vol.35, No.3, pp.430–448 (online), DOI: 10.1109/TSE.2008.71 (2009).

[15] Steven, P. and Reiss, G.E.: From the Concrete to the Abstract: Visual Representations of Program Execution, DMS 2005 (2005).

[16] Dan Pilone, N.P.: UML2.0 クイックリファレンス，オライリー・ジャパン (2006).

[17] Eclipse: Eclipse Oxygen, The Eclipse Foundation (online), available from ⟨https://eclipse.org/⟩ (accessed 2017-08-01).

© 2018 Information Processing Society of Japan

佐藤 拓弥

2016 年宮崎大学工学部情報システム工学科卒業．現在，宮崎大学大学院工学研究科工学専攻機械・情報系コース在籍．デバッグの効率向上やプログラムの可視化に関する研究に従事．

片山 徹郎　（正会員）

1991 年九州大学工学部情報工学科卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1995 年同大学院工学研究科情報工学専攻博士後期課程修了．同年奈良先端科学技術大学院大学情報科学研究科助手．2000 年宮崎大学工学部情報システム工学科助教授．2007 年より同准教授．ソフトウェア工学，特にソフトウェアのテスト技法や信頼性に関する研究に従事．博士（工学）．電子情報通信学会，日本ソフトウェア科学会各会員．

喜多 義弘　（正会員）

2004 年宮崎大学工学部情報システム工学科卒業．2006 年宮崎大学大学院工学研究科情報工学専攻博士前期課程修了．2010 年宮崎大学大学院工学研究科システム工学専攻博士後期課程単位取得満期退学．2012 年神奈川工科大学セキュリティ研究センター特別研究員．2015 年東京工科大学コンピュータサイエンス学部助教．ソフトウェアテストに関する研究に従事．博士（工学）．

山場 久昭　（正会員）

1988 年東京工業大学工学部化学工学科卒．1990 年東京工業大学大学院総合理工学研究科化学環境工学専攻修士課程修了．同年花王（株）入社．1993 年より宮崎大学工学部助手．2007 年より同助教．2010 年宮崎大学大学院博士後期課程システム工学専攻単位取得満期退学．生産システム設計・運用の計算機支援に関する研究に従事．博士（工学）．化学工学会，人工知能学会，計測自動制御学会各会員．

油田 健太郎　（正会員）

2003 年宮崎大学工学部情報システム工学科卒業．2005 年宮崎大学大学院工学研究科情報工学専攻博士前期課程修了．2006 年熊本県立大学総合管理学部助手．2009 年宮崎大学大学院工学研究科システム工学専攻博士後期課程修了．同年大分工業高等専門学校助教．2012 年同講師．2017 年より宮崎大学工学部情報システム工学科准教授．コンピュータネットワークに関する研究に従事．博士（工学）．電子情報通信学会，電気学会各会員．

岡崎 直宣　（正会員）

1986 年東北大学工学部通信工学科卒．1991 年東北大学大学院工学研究科電気および通信工学専攻博士後期課程了．同年，三菱電機株式会社入社．2002 年宮崎大学工学部助教授．2007 年同准教授を経て 2011 年より宮崎大学工学教育研究部教授．通信プロトコル設計，ネットワーク管理，ネットワークセキュリティ，モバイルネットワーク等の研究に従事．博士（工学）．電子情報通信学会，IEEE 各会員．

© 2018 Information Processing Society of Japan