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

中村 紘人[1] 片山 徹郎[1] 喜多 義弘[2] 山場 久昭[1] 岡崎 直宣[1]

概要：本研究では，プログラムのバグの原因特定を支援しデバッグの効率を向上させるために，Java プログラム実行時のデータ遷移可視化を行う手法を提案する．本手法はプログラム実行時の各変数の状態や更新といったデータ遷移を可視化することによって，プログラムの挙動の把握を支援し，デバッグを容易にする．我々は本手法の有用性を示すために，本手法を用いた可視化ツール TVIS(transitions visualization) を実装した．TVIS により，プログラマはバグを含んだプログラムの特異な挙動を一目で判断することが可能となり，Java プログラムでのバグの原因特定に要する時間を 46%減らすことができた．

1. はじめに

ソフトウェア開発におけるプログラムのデバッグ工程において，バグの原因特定には多くの時間が必要である [1]．プログラムのバグの原因を特定するためにはプログラム実行時における処理の流れやデータの遷移といったプログラムの挙動を十分に把握する必要がある．しかし，バグを含んだプログラムはプログラマが予期しない挙動を取る．そのため，バグを含んだプログラムの挙動を正しく把握することは難しく，バグの原因特定に多くの時間を必要とする．

バグの原因特定を支援する従来の手法としてシンスライシング [2] が存在する．シンスライシングは，プログラムスライシング手法 [3] の一種である．プログラムスライシングには，抽出するデータが多くなり，重要な情報が分かり難くなりやすいという問題がある．シンスライシングは，データの抽出をプログラムのデータ遷移に限定することで，必要最小限のデータを用いてデータ遷移の解析を可能とする．しかし，データ抽出の基準の選び方によっては，必要な情報を円滑に得られない場合がある．また各変数の更新のタイミングは示されないため，バグの原因特定に至るための充分な情報を得られるとは言えない．

そこで本研究では，Java プログラムのデバッグ効率を高めるために，バグの原因特定を支援するデータ遷移の可視化手法を提案する．本手法はプログラムのデータ遷移を可視化し，データ遷移の把握を容易にすることにより，プログラムの挙動の把握を支援する．バグを含んだプログラムの挙動の把握が容易になれば，効果的にバグの原因を特定できる．

提案手法の有用性を示すために，デバッグ支援ツール TVIS(transitions visualization) を試作し，その機能としてデータ遷移の可視化を実現した．TVIS の主要な機能は，データ遷移図，更新履歴表，スライシング機能である．特にデータ遷移図は，TVIS の最も特徴的な機能であり，プログラムのデータ遷移を表形式で示す．これらの TVIS のデータ遷移可視化機能は，バグを含んだプログラムの特異な挙動やデータ遷移を視覚的にプログラマに示す．その結果，プログラムの挙動の把握が容易になり，バグの原因特定を支援できる．

2. データ遷移

本研究におけるデータ遷移とは，プログラムの各変数がプログラムの実行時にいつ，どこで，どのような値に更新されたのかという変数の更新の流れのことを指す．プログラムのデータ遷移を把握することは，プログラム実行時における任意のタイミングでの各変数の状態を予想できることを意味する．そのため，データ遷移を把握できれば，プログラムの挙動の把握が容易になる．バグを含んだプログラムのデータ遷移を把握し，その挙動を正しく予想することが可能ならば，プログラマが望む正しいプログラムの挙動と，実際の挙動の差異を比較できるため，効果的にバグの原因を特定できる．

そこで，本研究ではデータ遷移を可視化してプログラマに示すことで，バグの原因特定を支援する．プログラマはデータ遷移の可視化により，バグを含むプログラムで

[1] 宮崎大学
University of Miyazaki
[2] 神奈川工科大学
Kanagawa Institute of Technology

© 2014 Information Processing Society of Japan

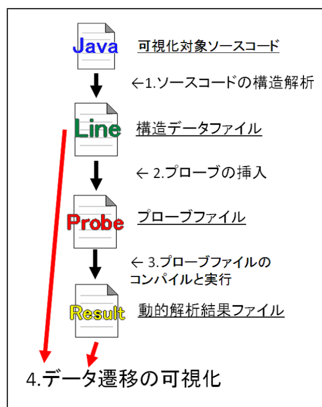

図 1 データ遷移の解析の流れ
Fig. 1 The flow of analyzing data transitions.

あっても，そのデータ遷移を容易に把握できるようになる．データ遷移を把握することによって，プログラムの挙動の把握も容易になり，効率よくバグの原因特定を行うことができる．

3. TVIS

本研究では提案した手法によるデータ遷移の可視化の有用性を示すために，デバッグ支援ツール TVIS を開発し，その機能としてデータ遷移の可視化を実現した．以下に TVIS によるデータ遷移の可視化について述べる．

3.1 データ遷移解析

TVIS によるデータ遷移の解析は，可視化対象のプログラムのソースコードに動的解析のためのプローブを挿入し，プログラム実行時における各変数のデータ遷移の情報をファイルに出力する．その情報を元にデータ遷移を解析し，可視化を行う．プローブの適切な挿入位置を判定するために，可視化対象のソースコードの構造を解析する必要がある．また，このソースコードの構造情報は可視化時に適切な図を生成するためにも必要になる．TVIS がデータ遷移の解析を行いデータ遷移を可視化するまでの流れを，**図 1** に示す．以下，各処理について説明する．

3.1.1 ソースコードの構造解析

可視化対象のプログラムから，構造データファイルを生成する．**図 2** に，構造データファイルの例を示す．

構造データファイルには，左から行番号，構文の識別番号，インデントの深さ，宣言された変数の情報，更新された変数名，更新に関わった変数名，整形したソースコードといった要素を保持する．これらの情報は，プログラムの実行時におけるデータ遷移を取得するために埋め込むプローブの挿入場所の判定基準，および可視化した図を適切に表示するために用いる．また，可視化の手法上，1 行に複数の構文がある場合等，ソースコードの書き方によっては可視化に問題がでる．そのため，行の分割等の，ソースコードの整形を行う．

構造データファイルは，可視化対象のソースコードを構文解析し，その結果から必要な情報を求め，ファイルに出力することで生成する．なお，構文解析には，JavaCC(Java Compiler Compiler)[4] で作成した JavaParser を用いた．

3.1.2 プローブの挿入

構造データファイルを用いて，ソースコードにプローブを挿入し，プローブファイルを生成する．プローブファイルとは，データ遷移を動的解析するために，ソースコードの適切な位置にプローブを挿入したファイルである．

図 3 に，プローブファイルの例を示す．図 3 において文の先頭が，（/*TVIS*/）になっている文が挿入したプローブである．プローブが出力する情報は，各変数がいつ，どのような値に更新されたか，その更新に関わったのは，どの変数であったのかといった変数のデータ遷移の情報である．

3.1.3 プローブファイルのコンパイルと実行

プローブファイルをコンパイルし実行することにより，動的解析結果ファイルを出力する．動的解析結果ファイルとは，プローブファイルに埋め込んだプローブにより出力したデータ遷移の情報を保持したファイルである．

図 4 に，動的解析結果ファイルの例を示す．図 4 に示すように，動的解析結果ファイルには，変数の更新の情報を出力する．

3.1.4 データ遷移の可視化

TVIS によるデータ遷移の可視化は，構造データファイルと動的解析結果ファイルを用いて行う．次節において，TVIS の各可視化機能について説明する．

3.2 データ遷移可視化

TVIS の主な機能は，データ遷移図，更新履歴表，スライシング機能である．**図 5** に，TVIS の画面例を示す．画面右上の表がデータ遷移図，緑の領域が更新履歴表，青い領域がスライシング機能により抽出したスライスのリストである．TVIS はこれらの機能によってデータ遷移を可視化する．以下，TVIS の各機能について説明する．

3.2.1 データ遷移図

データ遷移図は，TVIS の最も特徴的な機能であり，プログラムの各変数の更新とそのタイミングを示す表である．図 5 に，データ遷移図の例を示す．

データ遷移図とは，縦軸をソースコードの行番号，横軸をループの周回としたデータの更新値の表である．データ遷移図を用いることで，変数の各更新が起きたのは，ソースコードのどの行なのか，もしくは，どのループのどの周回で起きたのかを判断することが可能である．

データ遷移図上に色が濃い長方形で示す領域は，各ループの各周回を表している．この領域を以後，ループ領域と呼ぶ．ループ領域の色の濃さは対応するループの多重度を示し，色が濃いループ領域は，色が薄いループ領域に対応

```
6  #2 #5 #j/95/0        #j/          #i/j/          # for ( int ^*U = ^*I + 1 ; j < arr . length ; ^*I ++ )#
7  #0 #5 #              #            #              # {#
8  #1 #6 #              #            #              # if ( arr [ min ] > arr [ j ] )#
9  #0 #7 #              #            #              # {#
10 #0 #8 #              #min/        #j/            # ^*U = ^*I ;#
11 #0 #7 #              #            #              # }#
```

図 2　構造データファイルの例

Fig. 2　An example of the structure data file.

```
for ( int j = i + 1 ; j < arr . length ; j ++ )
{
/*TVIS*/TVIS_FILES.println("6#3#0#");
/*TVIS*/TVIS_FILES.print("6#2#"+"j"+"/"+"#"+j+"#");
/*TVIS*/TVIS_FILES.println("i"+"/"+"j"+"/"+"#");
        if ( arr [ min ] > arr [ j ] )
        {
                        min = j ;
                        /*TVIS*/TVIS_FILES.print("10#2#"+"min"+"/"+"#"+min+"#");
                        /*TVIS*/TVIS_FILES.println("j"+"/"+"#");
```

図 3　プローブファイルの例

Fig. 3　An example of the probe file.

```
6#3#0#
6#2#j/#1#i/j/#
6#3#1#
6#3#0#
6#2#j/#2#i/j/#
10#2#min/#2#j/#
6#3#1#
6#3#0#
```

図 4　動的解析結果ファイルの例

Fig. 4　An example of the result file of dynamic analysis.

するループの入れ子になったループを表現している．ループ領域の左上の起点の高さが同じループ領域は，同じループの周回であることを表す．例として，図 5 の赤い四角で囲んだ部分は，*Loop*0 の 3 度目の繰り返しの中で，*Loop*1 が 2 回繰り返したことを表現している．

なお，for 文等で，カウンタ変数を更新した結果，継続条件を満たさずループから脱出した場合に，そのカウンタ変数のスコープがループの中だけであるのならば，カウンタ変数の最後の更新はループの後に参照されないためデータ遷移図上では省略している．

データ遷移図上の値とループ領域から，変数の各更新がいつ行われ，どのような値に更新されたかを把握できる．例として，図 5 で青い領域で囲んだ更新は，*Loop*0 の 2 週目かつ，その中で実行された *Loop*1 の 2 週目の処理で，変数 *min* を値 3 に更新したということを示している．

データ遷移図の表示を表形式にすることにより，全体の更新の回数や，処理の流れを俯瞰できる．さらに，プログラムの実行時におけるデータ遷移の特異な挙動を見つけた時に，他のデータの状況を確認することも容易である．

3.2.2　データ遷移線

データ遷移線は，データ遷移図上で矢印として示し，変数の更新同士の関係を表す．この関係は，ある変数の値の生成に使用したオペランドは，どの変数のいつの更新で生成された値であるのかを意味する．データ遷移線の表示は，データ遷移図の値をクリックして基準を指定することで行う．

データ遷移線の例を，図 5 に示す．図 5 のデータ遷移図上に描画した赤い矢印が，データ遷移線である．なお，視覚的に分かり易くするために，TVIS は指定した値を青色で，関連する値を赤色で，それぞれ描画する．

このデータ遷移線を用いることにより，変数の更新により，プログラマが予期しない不審な値が生じた際に，その更新に関係する値を調べることが容易になる．そのため，不審な値の発生原因を特定することが容易になる．

3.2.3　更新履歴表

更新履歴表とは，各変数がプログラムの実行中に何度更新され，その都度，どのような値になったかを示す表である．更新履歴表はデータ遷移を各変数別にまとめているという点で，データ遷移図と異なる．更新履歴表の例を，図 5 右下の緑の領域に示す．

更新履歴表の縦軸は各変数名，横軸は更新回数であり，表の値は変数の各更新の後の値である．また N で表す列は，各変数がソースコード上で宣言された行を示す．$Type$ で表す列は各変数のデータ型の頭文字を示す．

更新履歴表からは，各変数が異常な値に更新されていないか，更新回数に異常はないかといった情報を容易に得ることができる．

3.2.4　スライシング

補助的な機能として，各変数の任意の状態へスライシングを行うことができる．データ遷移図もしくは更新履歴表の任意の値をクリックすると，その状態へのスライシングを実行する．スライシングの結果は，画面の青い領域に表示する．スライシングの結果表示の例を，図 5 左下の青い領域に示す．

TVIS のスライシング機能は，データ遷移図と更新履歴表を併用することにより，プログラム全体のデータ遷移を俯瞰しつつスライシングの基準を定めることができるため，より効率的にスライシングを行うことが可能である．

4.　適用例

実際に TVIS を用いて，バグを含んだプログラムの可視化を行い，TVIS の有用性の確認を行う．例に用いるのは，Java 言語で作成された一般的なバブルソートプログラムである．TVIS がこのプログラムを可視化した図を，**図 6** に示す．

図 6 のプログラムは，ループ条件に欠陥がある．図 6 上

© 2014 Information Processing Society of Japan

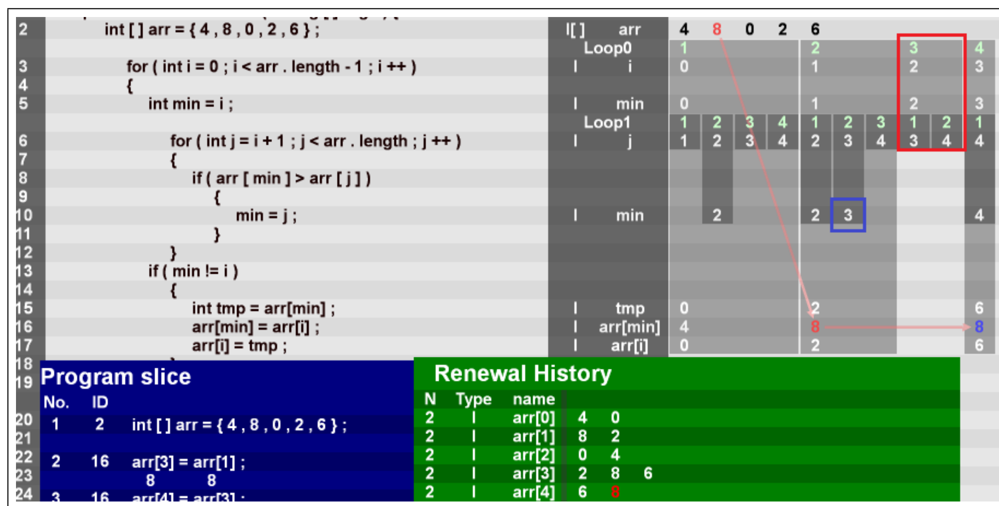

図 5　TVIS の画面例

Fig. 5　An example of the screen of TVIS.

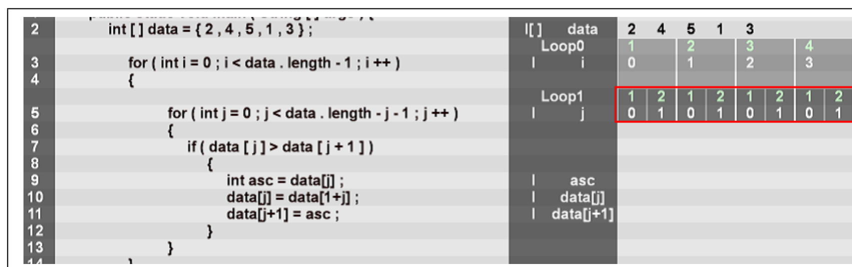

図 6　バグを含んだプログラムの適用例

Fig. 6　An example of application to a program including bugs.

で，行番号 5 の for 文の条件は（$j < data.length - j - 1$）ではなく（$j < data.length - i - 1$）が正しい．このような名前が似ている変数の混同は，よくある欠陥であるとともに，デバッグ時に見落としやすい欠陥である．この欠陥のために，このプログラムは，ソート前と全く変わらない順番のまま配列を返してしまう．

図 6 に示すデータ遷移図からは，$Loop1$(行番号 5 から始まる for 文) と行番号 9〜11 の配列要素の入れ替え処理の異常な挙動が一目でわかる．図 6 で示す赤色の四角で囲んだ部分に着目すると，$Loop1$ は 4 度，実行されているが，その全てにおいて 2 回しか繰り返しを行っていない．さらに $Loop1$ のループ用カウンタ変数 j の値が 2 以上になっていないことから，配列の後半の要素が全く比較されていないことがわかる．

以上，TVIS によるデータ遷移の可視化によって，バグを含んだプログラムの挙動の把握に役立つ情報を得られることを確認できた．

5.　評価

今回提案した手法によるデータ遷移の可視化を，デバッグ支援ツール TVIS として実装した．また，実際にバグを含んだプログラムのデータ遷移を可視化し，本手法の有用性の確認を行った．適用例において，本手法によるデータ遷移の可視化は，バグを含んでいるためにプログラマの予期しない挙動を取るプログラムに対して，その挙動の把握を支援できることを示した．以下では，被験者を用いた TVIS の評価実験を行う．また本手法と従来の手法を比較し，本手法の有用性について評価する．

5.1　評価実験

実際に TVIS がバグの原因特定に貢献できていることを確認するために，被験者を用いた評価実験を行う．なお，評価実験に参加する被験者は，情報系学科に所属し，Java と統合開発環境 Eclipse を使用した経験のある学部 4 年生及び大学院生である．今回の評価実験に用いるプログラムの主要な部分を，**図 7** に示す．

図 7 は，int 型配列に入っている整数値の平均値，最小値，最大値をそれぞれ計算し出力するプログラムである．このプログラムは 2 つの欠陥を含んでおり，被験者がその両方の欠陥を特定できるまでの時間を計測する．なお，図 7 が含む欠陥は，1 つは 10 行目で Int 型変数同士の割り算の結果を $double$ 型変数に代入していることである．もう 1 つは，変数 min の初期化である．変数 min を配列要素と順に比較し，min より小さければ，比較対象を min に代入する．これを配列要素の数だけ繰り返すことで最小値の算出を行うが，4 行目で宣言した min を 0 で初期化した

© 2014 Information Processing Society of Japan

```
1          int[] sco={6,2,7,9,1,8};
2          int sum=0;
3          double ave=0;
4          int min=0;
5          int max=0;
6          int num=sco.length;
7          for(int i=0;i<num;i++){
8              sum=sum+sco[i];
9          }
10         ave=sum/num;
11         for(int i=0;i<num;i++){
12              if(min > sco[i]){
13                    min=sco[i];
14              }
15         }
16         for(int i=0;i<num;i++){
17              if(max < sco[i]){
18                    max=sco[i];
19              }
20         }
```

図 7 評価実験用プログラムの主要部分
Fig. 7 The program for evaluation experiment.

ままであるので，この 0 が最小値になってしまっている．

被験者を TVIS を使用する被験者と，TVIS を使用しない被験者の 2 グループに分け，それぞれが必要とした時間を比較する．どちらの被験者にも，事前に図 7 の処理内容，平均値と最小値の出力が間違っており，その原因が 2 つ存在することは伝えてある．TVIS を使用する被験者には，アルゴリズム，バグともに関係性のない例題用プログラムを用いて，事前に TVIS の操作説明を行っておく．TVIS を使用しない被験者は，Eclipse 上でバグの原因特定を行った．その際，Eclipse のデバッグ機能の使用，もしくは *print* 文によるプローブ挿入等のバグの原因を特定するための手段は制限せず自由に行わせた．

評価実験の結果を，**表 1** に示す．TVIS を使用した被験者は TVIS を使用しなかった被験者よりも平均して約 46%，所要時間が短い．

評価実験において，TVIS を使用しなかった被験者は手作業でのいくつものプローブの挿入，もしくはループの中にブレークポイントを設定し，何度も実行と中断を繰り返すといった面倒な作業を行いバグの原因を特定した．それに対し，TVIS を使用した被験者は上述のような面倒な作業をすることなくバグの原因を特定することができたため，より短い時間でバグの原因を特定できている．このことから，TVIS のデバッグ支援により，デバッグの効率が向上していると言える．

しかし，TVIS を使用した被験者は，TVIS をうまく操作できず手間取ってしまった．この手間取りは，TVIS の操作を行い，その出力を確認していくことで解消されたため，その悪影響は評価実験の開始直後に限られている．この原

表 1 評価実験の測定結果
Table 1 The result of evaluation experiment.

	TVIS 使用
被験者 A	168 秒
被験者 B	216 秒
被験者 C	139 秒
被験者 D	189 秒
平均	178 秒

	TVIS 未使用
被験者 E	411 秒
被験者 F	251 秒
被験者 G	321 秒
被験者 H	352 秒
平均	333 秒

因は，何度も使ったことがある Eclipse のインターフェースに対し，TVIS のインターフェースは彼らにとって馴染みのないものであるとともに，直観的に操作することが難しかったためであった．このことから，TVIS のインターフェースを直観的に操作しやすいものに改良する必要があると考えている．

5.2 従来研究との比較

動的スライシング [5] は，指定した状態の生成に関係した処理をスライスとして抜き出すことで，プログラムの挙動の解析を行う手法である．しかし，スライスに含まれなかった部分の情報は示されない．そのため，スライシングの基準の選び方が適切でなければ，有用な情報は得られず，場合によっては，何度もスライシングの基準を設定し直す必要がある．さらに，プログラムの実行結果のみからスライシングの基準を設定しなければならず，非効率的である．本手法では，データ遷移図と更新履歴表を併用することによって，途中の状態へのスライシングも容易に行うことができるため，効率的にデータ遷移を解析できる．また，データ遷移図では，更新のタイミングといった情報を得ることも可能である．

ブレイクポイントはもっとも多用されるデバッグ手法の 1 つである [6]．ブレイクポイントは，基準の設定に相応の経験が必要になるという問題があるが，ブレイクポイントを自動生成する手法 [7] もあり，バグの原因特定に有効な手法である．しかし，プログラムの実行時における処理の全体のある一点だけの情報では，バグ発生時の状況を正しく把握するのは難しく，別の場所での再解析を行うなど情報の追加が必要になってくる場合が多い．本手法では，プログラマが予期しない不審な処理を見つけた時に，その処理に関連した変数やループに関する情報をデータ遷移図等から取得可能であり，よりプログラムの実行時における挙動を把握し易い．ただし，本手法は図の情報量が多くなりやすいため，処理が多いプログラムでの有用性はブレイクポイントよりも損なわれ易い．

高い抽象度の図で大規模なプログラムの可視化を行うツール [8] やマルチスレッドプログラムに対応できるツール [9] と比べると，本手法に適用可能なプログラムの範囲は狭い．可視化は適度な大きさのプログラムでさえも巨大化する問題があり [10]，本手法のように図の抽象度が低い

© 2014 Information Processing Society of Japan

場合，生成した図が巨大化しやすく，適用可能なプログラムの規模が小さくなる．データ遷移図が巨大化した場合，画面のスクロールを行うことはできるが，一画面にデータ遷移図の一部分しか表示できないため，データ遷移図から変数同士の関係や特異な挙動を読み取りにくくなる．そのため，本手法の有効な可視化の範囲は，データ遷移図の大部分が一画面に入る大きさである．本手法はソースコードの行数が 1 万行を超える，もしくはループの回数が 1 万回を超える場合も可視化自体は可能であるが，有用性が高いのは，ループ回数が高々 100 回程度に収まるプログラムまでと考えられる．

また，本手法は現段階では，単一のモジュールのみ対応である上，マルチスレッドとプリミティブ型以外のデータには対応していない．これらのことを踏まえると，適用可能なプログラムの幅が狭いことが，本手法の問題点の 1 つである．

この問題点を解決するためには，図の巨大化への対策を行う必要がある．図の巨大化を緩和することによって，より処理の多いプログラムを適応できると考えられる．また，本手法が上述した構文に対応していない理由は，図の巨大化を引き起こしやすく，本手法での可視化が難しいためである．そのため，図の巨大化を緩和することによって，非対応の構文への対応も可能になってくると考えられる．図の巨大化の緩和手段として，可視化の範囲や，抽象度を調整可能にし，可視化の度合いを変更できるようにすることによって，巨大な図を見やすくすることを検討している．

6. 結論

本研究では，Java プログラムのデバッグ効率を高めることを目的とし，Java プログラム実行時のデータ遷移可視化手法を提案した．本手法は，予想しにくいデータ遷移を可視化することで，プログラムの挙動を容易にし，バグの原因特定を助ける．提案した手法の有用性を示すために，デバッグ支援ツール TVIS を開発し，プログラムのデータ遷移の可視化を実現した．

本稿では，実際に TVIS を用いて，バグを含んだプログラムのデータ遷移を可視化した．その結果，バグを含みプログラマの予期しない挙動を取るプログラムの挙動の把握を支援できることを示した．プログラムの挙動を把握しやすくなったことにより，効果的なバグの原因特定が可能になった．そして，被験者を用いて TVIS でデバッグを行い，デバッグの効率が 46%向上できたことを示した．

本手法によるデータ遷移の可視化は，従来のデバッグ支援手法では得られなかった情報を得ることが可能である．特にデータ遷移図は，処理の量の異常や，データの更新回数の異常など特異な挙動を視覚的に表すことができる．さらに，特異な挙動を見つけた際に，他の変数の状況を知ることも容易である．

以上のことから，本研究が提案したデータ遷移の可視化手法は，バグの原因特定を支援でき，Java プログラムのデバッグ効率を高めることが可能である．本手法は，Java プログラムの実行時におけるデータ遷移の把握を容易にし，プログラマがプログラムの挙動を正しく把握することを支援する．バグを含んだプログラムの挙動を正しく理解することで，バグの原因特定を効果的に行うことができるため，デバッグの効率を高めることが可能である．

以下に今後の課題を示す．

- インターフェースの改良
  評価実験の結果，被験者が TVIS をうまく操作できていないことが分かった．これは，TVIS のインターフェースが被験者らにとって馴染みのないものであるとともに，直観的に操作することが難しかったためであった．そのため，インターフェースの改良により，バグの原因特定に要する時間をさらに削減できると考えられる．
- 図の巨大化への対応
  ステップ数が多い，もしくはループ回数が多いプログラムを可視化すると，図が巨大になり理解が困難になる．そのため，可視化の範囲や，抽象度を調整可能にし，可視化の度合いを変更できるようにすることによって，巨大な図を見やすくすることを検討している．

参考文献

[1] Pressman, R. S.: *Software Engineering A Practitioner's Approach*, McGraw-Hill Science, 5th edition (2001).

[2] Sridharan, M., Fink, S. J. and Bodik, R.: Thin slicing, *SIGPLAN Not.*, Vol. 42, No. 6, pp. 112–122 (2007).

[3] Weiser, M.: Programmers Use Slices When Debugging, *Comun. ACM*, Vol. 25, No. 7, pp. 446–452 (1982).

[4] Vanter, M. V. D.: JavaCC Home Page, java.net (online), available from ⟨http://javacc.java.net/⟩ (accessed 2014-05-13).

[5] Agrawal, H. and Horgan, J.: Dynamic Program Slicing, *SIGPLAN Not.*, Vol. 25, No. 6, pp. 246–265 (1990).

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

[7] Zhang, C., Yang, J., Yan, D., Yang, S. and Chen, Y.: Automated Breakpoint Generation for Debugging, *Journal of Software*, Vol. 8, No. 3, pp. 603–616 (2013).

[8] Reiss, S. P. and Eddon, G.: From the Concrete to the Abstract: Visual Representations of Program Execution, *DMS*, pp. 315–320 (2005).

[9] Lönnberg, J., Ben-Ari, M. and Malmi, L.: Java replay for dependence-based debugging, *PADTAD*, pp. 15–25 (2011).

[10] De Pauw, W., Lorenz, D., Vlissides, J. and Wegman, M.: Execution patterns in object-oriented visualization, *COOTS'98*, pp. 219–234 (1998).

© 2014 Information Processing Society of Japan