

データ遷移の可視化手法によるバグの原因特定支援について

中村紘人^{a)} ・ 片山徹郎^{b)}

Supporting to Find the Cause of a Bug by a Method of Visualizing Data Transitions

Hiroto NAKAMURA, Tetsuro KATAYAMA

Abstract

It takes much time to find the cause of a bug in debugging of programs. Finding the cause of a bug needs to comprehend a flow and data transitions in executing programs. It is difficult to grasp behavior in executing the programs whose behavior is unexpected by a bug. We propose a visualizing method of data transitions to support finding the cause of a bug for Java programs in order to improve efficiency of debugging. We have implemented TVIS(transitions visualization) in order to confirm efficiency of the proposed method. The data transitions diagram is the most characteristic function of TVIS which shows the data transitions in executing programs as a table. It can show visually abnormal behavior: no data renewed at all, data abnormally renewed, and so on. Because abnormal behavior is detected in the data transitions diagram at first glance, it is useful for programmers in finding the cause of a bug. Hence, the method can support to find the cause of a bug.

Keywords: Programming, Java, Visualization, Debug, Dynamic analysis

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の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の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、バグの原因となる箇所を探すために多くの時間を必要とする。特に経験が乏しく、プログラムの挙動を予想する能力が低いプログラマにとっては多大な時間を費やす作業になる。

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易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スライシングの手法である。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十分な情報を得られるとは言えない。

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することが可能になる。

本研究では、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と更新履歴表と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プログラマに示し、プログラムの挙動の把握を容易にすることにより、バグの原因特定を支援できる。

本論文では、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の可視化を行い、本手法がバグの原因特定を支援できることを示す。

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

a)情報システム工学専攻大学院生

b)情報システム工学科准教授

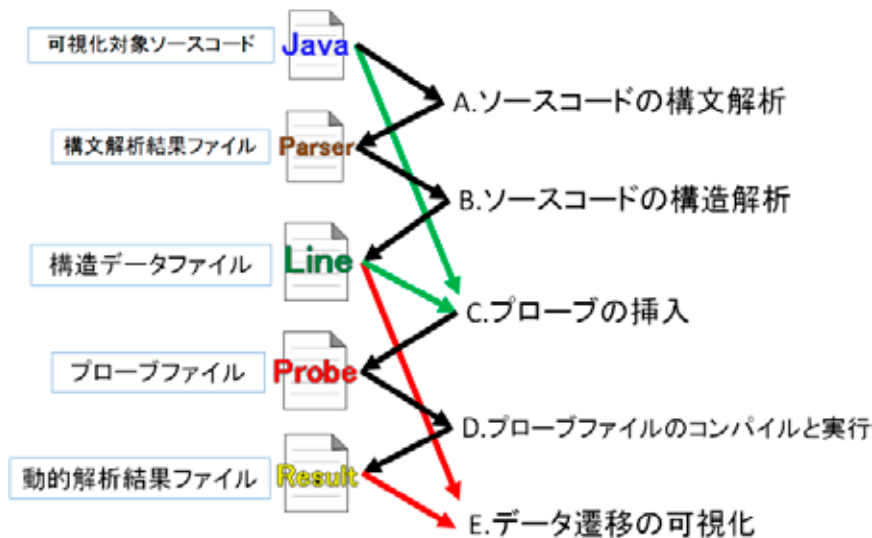


図1. データ遷移可視化の流れ。

プログラムの実行時にいつ、どのような値に更新されていっ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、プログラムの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、プログラムの実行時における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、実行時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調べることが可能になり、効果的にバグの原因を特定できる。

データ遷移の把握は、バグの原因を特定する上で重要である。しかし、バグを含んだプログラムはプログラマが期待する挙動を取らず、さらにプログラマはバグの原因を特定できていない。このような状況で、各変数のデータ遷移を把握することは困難である。プログラマは、プログラムのどこまでが正しいかを判断できず、バグの原因特定に多くの時間を費やすことになる。

すなわち、実際にはデータ遷移を正しく把握し、バグの原因特定を効果的に行うことは難しい。

3. TVIS

本研究では、プログラムのデータ遷移を可視化することにより、プログラマのデータ遷移の把握を支援する。プログラマがデータ遷移を把握し易くすることにより、プログラムの実行時における挙動の把握を助け、バグの原因特定の効率を高める。

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を試作し、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 データ遷移の解析

データ遷移の可視化を行うためには、可視化対象のプログラムに動的解析を行いプログラムの実行時におけるデータ遷移を解析する必要がある。

そこで、最初にTVISによるデータ遷移の解析について説明する。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、プログラムの実行時における各変数のデータ遷移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。そのため、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図を適切に生成するためにも必要である。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。また各処理について以下に説明する。

A. ソースコードの構文解析

可視化対象のプログラムのソースコードを構文解析し、構文解析結果ファイルを生成する。

図2に、構文解析結果ファイルの例を示す。図2に示すように、構文解析結果ファイルには、ソースコードをコンパイルが可能であるプログラムの文法の最小単位であるトークンと、そのトークンごとに割り当てた固有のIDに分割して出力する。トークンとIDのデータの区切りは「#」記号を用いて行う。なお、本研究では「#」記号を一貫してデータの区切りとして扱う。この構文解析には、JavaCC(Java Compiler Compiler)⁴⁾で作成したJavaParserを用いて実装した構文解析器を用いる。

```
650#1
841#;
333#i
490#++
825#)
746#[
820#for
824#(
95#int
213#j
211# =
650#0
831#;
```

図2. 構文解析結果ファイル。

```

21#
0 #4 #0 # # # # public class Sample {#
1 #7 #1 # # # # public static void main ( String [ ] args ) {#
2 #9 #2 #data/95/1 #data/ # # # # int [ ] ^xU = { 4 , 7 , 10 , 2 , 5 } ;#
3 #2 #3 #i/95/0 #i/ # # # # for ( int ^xU = 0 ; i < data . length - 1 ; ^xI ++ )#
4 #0 #3 # # # # {#
5 #2 #5 #j/95/0 #j/ # # # # for ( int ^xU = 0 ; j < data . length - i - 1 ; ^xI ++ )#
6 #0 #5 # # # # {#
7 #1 #6 # # # # if ( data [ j ] > data [ j + 1 ] )#
8 #0 #7 # # # # {#
9 #0 #8 #asc/95/0 #asc/ #data[j]/ #data[j+1]/ # # # # int ^xU = ^xI ;#
10 #0 #8 # #data[j]/ #data[j+1]/ # # # # ^xU = ^xI ;#
11 #0 #8 # #data[j+1]/ #asc/ # # # # ^xU = ^xI ;#
12 #0 #7 # # # # }#
13 #0 #5 # # # # }#
14 #0 #3 # # # # }#
15 #2 #3 #i/95/0 #i/ # # # # for ( int ^xU = 0 ; i < data . length ; ^xI ++ )#
16 #0 #3 # # # # {#
17 #0 #4 # # # # System . out . print ( data [ i ] + " " ) ;#
18 #0 #3 # # # # }#
19 #8 #1 # # # # }#
20 #10#0 # # # # }#

```

図 3. 構造データファイル.

B. ソースコードの構造解析

構文解析結果ファイルを用いて、構造データファイルを生成する。構造データファイルは、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、および可視化を行う際に図の形を整えるために用いる。

図3に、構造データファイルの例を示す。図3に示すように、構造データファイルには、左から、行番号、命令の識別番号、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、命令文といった要素を出力する。なお、元のソースコードの文を可視化するために整形しているため、インデントの深さは元のソースコードと異なる。命令文は元のソースコードから、更新された変数名と更新に関わった変数名の部分に識別記号を入れ替えている。これらの情報により、各変数がいつ更新され、どのような値になったのか、そしてその更新にはどのような変数が関係したのかといったデータ遷移に関わる命令を識別する情報を取得できる。また、同じ名前の変数が複数存在する場合は、混同しないように各変数のスコープ領域を計算し、別の変数として扱うように管理する。

C. 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、動的解析によりデータ遷移を解析するために、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4に、プローブファイルの例を示す。図4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。またループ処理を解析するために、ループの開始と終

了のタイミング等を入力する。ループ処理を解析するのは、変数の各更新のタイミングをループ回数に関連付けて可視化するためである。ループ処理を解析するためには、ソースコード内のループの開始地点と終了地点の特定が必要である。構造データファイルが持つ命令の識別番号でループの開始地点を特定でき、ループの終了地点は、インデントの深さから特定できる。そのため、ループ処理の情報を出力するプローブを挿入することが可能になる。

D. 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入力する。動的解析結果ファイルは、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5に、動的解析結果ファイルの例を示す。図5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。これらの情報から、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情報を取得することができる。また、変数によって要素を指定された配列要素のように、静的解析の段階では実際にどの要素であるのかを判別できない要素については、実際にはどの要素であるのかを、この時点で確定する。さらに、変数の各更新のタイミングをループ回数に関連付けて可視化するために、ループの開始と終了のタイミングを出力する。

E. 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。

```

int [ ] data = { 4 , 7 , 10 , 2 , 5 };
/*TVIS*/TVIS_FILES.println("2#1#"+data+"#"+4, 7, 10, 2, 5, "+#");
for ( int i = 0 ; i < data . length - 1 ; i ++ )
{
/*TVIS*/TVIS_FILES.println("5#3#0#");
/*TVIS*/TVIS_FILES.print("3#2#"+i+"/"++"#"+i+"#");
/*TVIS*/TVIS_FILES.println(i+"/"++"#");
for ( int j = 0 ; j < data . length - i - 1 ; j ++ )
{
/*TVIS*/TVIS_FILES.println("5#3#0#");
/*TVIS*/TVIS_FILES.print("5#2#"+i+"/"++"#"+j+"#");
/*TVIS*/TVIS_FILES.println(j+"/"++"#");
if ( data [ j ] > data [ j + 1 ] )
{
int asc = data [ j ];
/*TVIS*/TVIS_FILES.print("9#2#"+asc+"/"++"#"+asc+"#");
/*TVIS*/TVIS_FILES.println("data"+["+(j)"+"]+"/"++"#");
data [ j ] = data [ j + 1 ];
/*TVIS*/TVIS_FILES.print("10#2#"+data+"["+i+"]+"/"++"#"+data [ j ]+"#");
/*TVIS*/TVIS_FILES.println("data"+["+(j+1)"+"]+"/"++"#");
data [ j + 1 ] = asc ;
/*TVIS*/TVIS_FILES.print("11#2#"+data+"["+i+"]+"/"++"#"+data [ j + 1 ]+"#");
/*TVIS*/TVIS_FILES.println("asc"+"/"+"#");
}
}
/*TVIS*/TVIS_FILES.println("5#3#1#");
}
/*TVIS*/TVIS_FILES.println("3#3#1#");
}

```

図 4. プローブファイル。

```

5#3#0#
5#2# j/#2#j/#
9#2# asc/#10#data[2]/#
10#2#data[2]/#2#data[3]/#
11#2#data[3]/#10#asc/#
5#3#1#
5#3#0#
5#2# j/#3#j/#
9#2# asc/#10#data[3]/#
10#2#data[3]/#5#data[4]/#
11#2#data[4]/#10#asc/#
5#3#1#

```

図 5. 動的解析結果ファイル。

3.2 データ遷移の可視化

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6に、TVISの画面例を示す。画面右上の表がデータ遷移図、画面右下のRenewal Historyウィンドウが更新履歴表、画面左下のProgram sliceウィンドウがスライシング機能により抽出したスライスの一覧である。TVISはこれらの機能によってデータ遷移を可視化する。以下に、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のもっと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7に、データ遷移図の例を示す。なお、図7は説明のために四角の枠線で図の要素を強調している。

データ遷移図は、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す。図7に示す例で、色が濃いループ領域は、色が薄いループ領域に対応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、同じループの周回を表す。図7のループ領域に着目すると、最初のfor文は4回繰り返す、2番目のfor文は、計10回繰り返した事がわかる。また、図7の枠A(%1)で囲んだ部分は、最初のfor文の2度目の繰り返しの途中で、2番目のfor文が3回繰り返したことを表現している。

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7の枠B(%2)で囲んだ更新に着目する。この更新に対応する行は、変数ascの更新を行う行である。また、最初のfor文の2個目のループ領域と、2番目のfor文の2個目のループ領域の中で更新されている。さらに更新後の値が7であるということは、変数ascが7に更新されたということを表している。よって、最初のfor文の2週目かつ、その中で実行された2番目のfor文の2週目の処理で、変数ascは値7に更新されたということがわかる。

このデータ遷移図は、各データの更新の流れを示すことができる。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

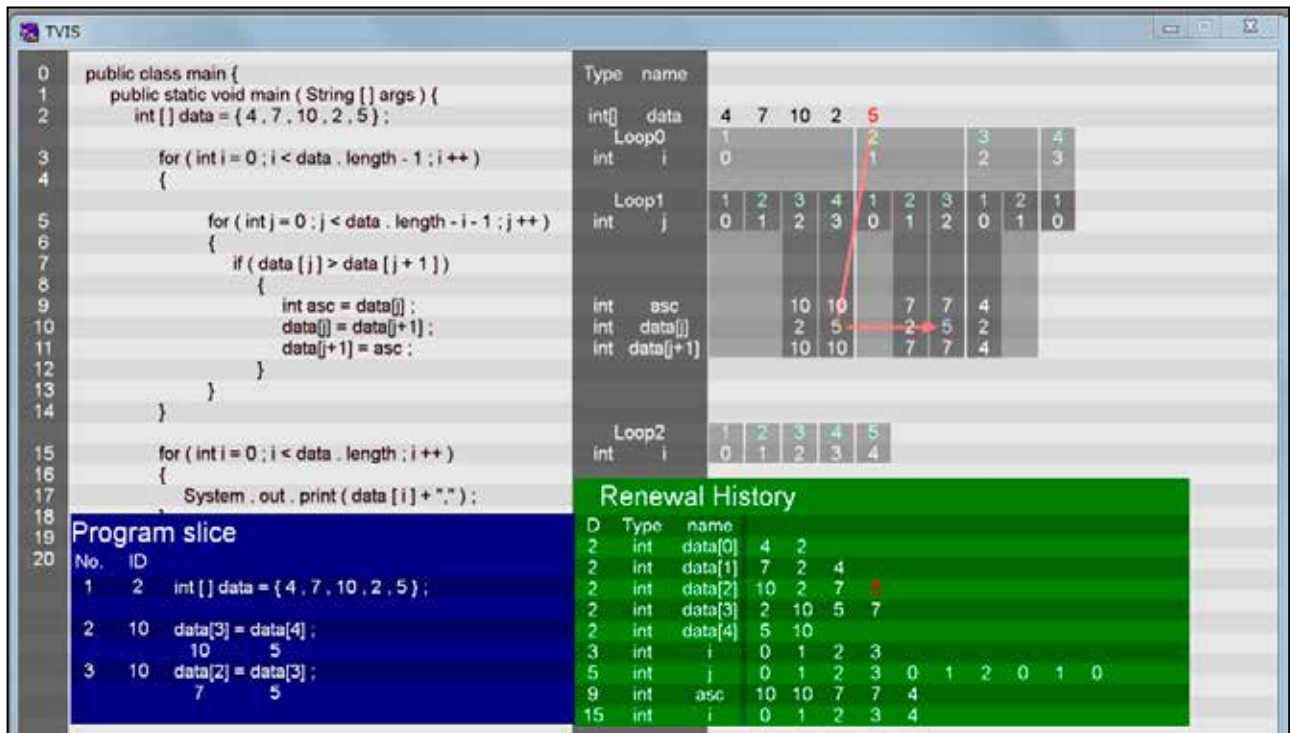


図 6. TVIS の画面例.

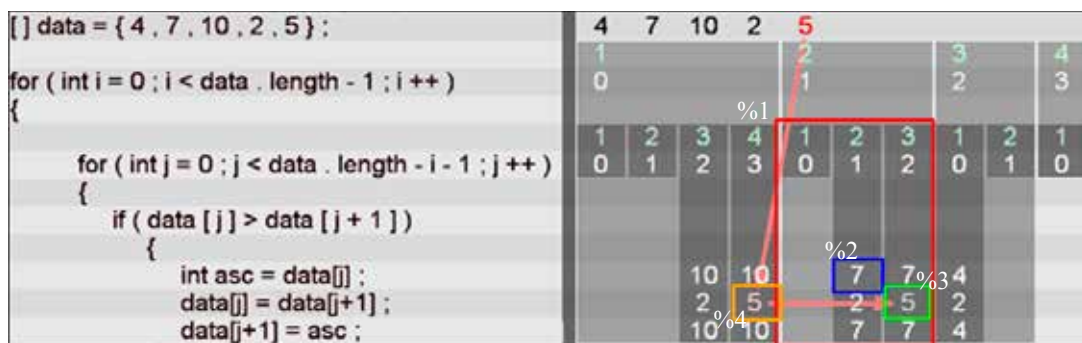


図 7. データ遷移図の例

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、ある変数の更新に使われたのは、どの変数のいつの状態なのかを意味する。データ遷移図の値をクリックして状態を指定することで、その状態になった更新に関わった変数の状態から矢印を描画する。

データ遷移線の例を、図7に示す。図7のデータ遷移図上に描画した矢印が、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描画している。図7のデータ遷移線は、枠C(%3)で囲んだ値を指定した際のものである。この時、枠D(%4)で囲んだ値から、枠Cで囲んだ値へデータ遷移線を描画している。このデータ遷移線は、枠Cで囲んだ値の生成に枠Dで囲んだ値が関係したことを示している。枠Cで囲んだ値の生成を行う命令は、`data`配列の要素に別の要素を代入する命令である。つまり、ここで代入に使われた配列要素の値が枠Dで囲んだ箇所生成された値である

ということを示している。

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、その更新に関わった変数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6右下のRenewal Historyウィンドウに示す。

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D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を示す。例として、図6に示す更

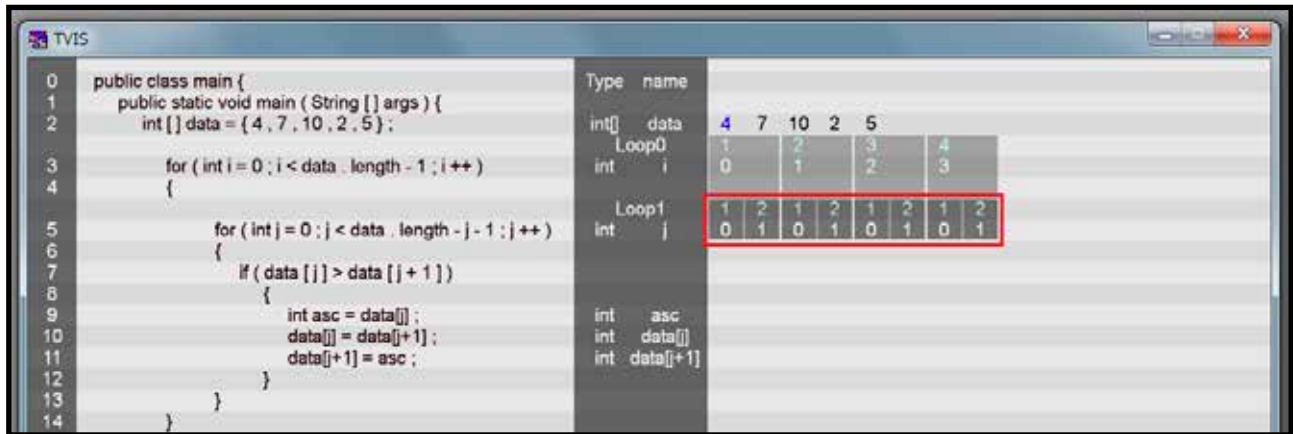


図 8. バグを含んだプログラムの適用例 A.

新履歴表のdata[3]の行に着目する。この行は、ソースコードの2行目で宣言されたint型の変数data[3]が4度更新され、最終的に値は7になったということを示している。また、図6での変数iの扱いからわかるように、同名の変数が複数ある場合、スコープ領域の違いから区別し、別の変数として管理する。

このように、更新履歴表は変数毎にデータ遷移を把握することが可能である。そのため、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シン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、その状態へのシン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。スライシング結果の表示の例を、図6左下のProgram sliceウィンドウに示す。

TVISのスライシング機能は、一度のクリックで行うことができる。さらに、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となる。各変数の最終的な状態だけではなく途中の状態をスライシングの基準に指定することも可能であるため、より重要な部分に限ったデータ遷移の解析を行うことが可能である。

TVISのスライシングの結果表示は、実際に更新に使われた変数名を表示する。図6で、スライシング結果の2行目は、「data[3] = data[4];」となっているが、実際のソースコード上では、「data[j] = data[j+1];」である。変数による要素指定を行っている場合、ソースコードをそのまま抽出するだけでは、実際にどの変数が関わっているのかが分かり難い。そこで、TVISは、このように実際に更新に使われた変数名を示すことでデータ遷移を把握し易くしている。

このように、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で、より効率的なデータ遷移の解析を可能とする。

4. 適用例

実際にTVISを用いて、バグを含んだ2種類の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を例に用意にした2種類のプログラムである。これらのプログラムは、それぞれ異なるバグを含んでいる。

図8に示すバブルソートプログラムは、ループ条件に間違いがある。図8上で、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、よくある間違いであるとともに、デバッグ時に見落としやすい間違いである。この間違いの結果、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8に示すデータ遷移図からはfor1(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8で示す枠で囲んだ部分に着目すると、for1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。また、for1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がわかる。for1の正しい挙動は、図6のように、ループ数が徐々に減っていくものである。そのため、for1の条件設定が疑わしいことがわかる。

図9に示すバブルソートプログラムは、ループ用カウンタ変数の間違いを含んでいる。図9上での行番号7から、行番号12まで使われている全ての変数iは変数jの間違いである。単に変数を混同しただけの間違いではあるが、複数の行が連続して間違っているため、気づきにくくなっている。この間違いの結果、このプログラムは、配列要素の一部をソートしただけで処理を終えてしまう。

図9に示すデータ遷移図からは、配列要素の入れ替え処

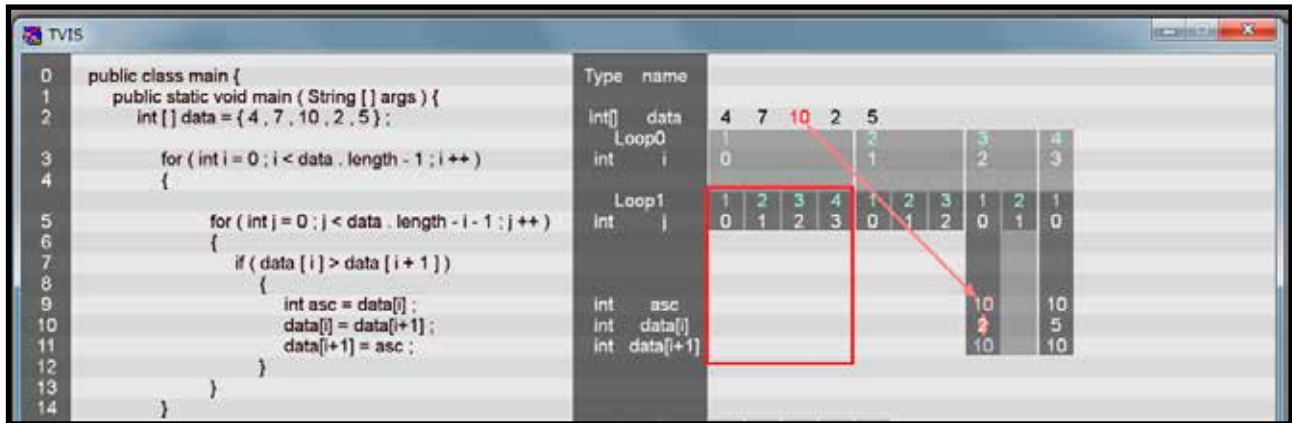


図9. バグを含んだプログラムの適用例 B.

理の挙動に異常があることがわかる。図9の枠で囲んだ部分に着目すると、for1は、for0(行番号3から始まるfor文)の最初のループの中で4回、繰り返している。そのため、行番号7のif文で全ての配列要素の比較を行っているはずだが、実際には配列要素の入れ替えが全く行われていない。そのため、行番号7のif文が疑わしいことがわかる。

以上、2つの可視化結果から、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を支援でき、バグの原因特定に有益な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ム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。よって、プログラムのバグの原因特定を行う上で、有益な情報を提供できるため、デバッグの効率を高めることができると言える。以下では、本手法によるデータ遷移の可視化と従来の手法との比較を行い、本手法の有用性について評価する。

シンスライシングを含めた動的スライシング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、プログラムの実行時における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。効果的にスライシングを行うには、別途にプローブを埋め込んで取得できる情報を増やす等、別の手法を併用しなければならず手間がかかる。また、図8に示すようなケースでは、どの

配列要素も初期化以外に更新がないため、どの配列要素をスライシングの基準に指定しても初期化処理以外のスライスが得られない。そのため、従来のスライシングでは有用な情報を得ることは難しい。本手法では図9のようなケースでも、データ遷移図から各更新やループの実行状況が分かるため、よりプログラムの挙動の把握に役立つ情報を提供できる機会が多くなる。また、スライシングのように解析の基準選びに時間をかける必要もないため、より効率的である。

デバッグで最も多用される手法の1つとしてブレイクポイントが挙げられる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。しかし、本手法は図の情報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。

高い図の抽象度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。また、本手法はマルチスレッドのプログラムを考慮していない。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、本手法の問題点となっていると判断できる。

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、

本手法がマルチスレッドプログラムに対応していないのは、複数のスレッドの処理を可視化しなければならないため図が非常に巨大化し易く、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れば、マルチスレッドプログラム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表示形式の改良と可視化の局所化を検討している。表示形式の改良は、図で変数の値を省略表示する等して、描画領域を小さく抑えることである。可視化の局所化は、可視化を行う箇所を限定することにより図の巨大化を抑えることである。これらのことを導入することにより、適用可能なプログラムの規模を大きくすることができる。

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した。また、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を試作して、プログラムのデータ遷移の可視化を実現した。そして、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ムの予期しない挙動を取るプログラムの挙動の把握の支援をできることを示した。そして、プログラムのバグの原因特定を行う上で、有益な情報を提供することが可能であることを示した。

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が容易である。

これら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、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。

● 情報の表現形式の改良

小数点型変数や文字列型変数は、各図で値を表示する際に、領域を大きく占領してしまう。そのため、これらの変数を多用するプログラムをTVISで可視化した場合、各図が巨大化してしまい図の利便性が損なわれる問題がある。

この問題に対しては、各変数の表現形式の改良による解決を検討している。図の領域を大きく占領する変数の値の

表示に省略表示や入れ子処理を導入することによって、図の領域を節約することが可能になると考えられる。

● 可視化の局所化の導入

現在、TVISがステップ数の多いプログラムを可視化すると、図が巨大になりすぎて理解しにくくなり、理解支援ツールとしての効果が期待できなくなる問題がある。

この問題に対しては、TVISがプログラムの特定の箇所のみを可視化できるようにすることによる解決を検討している。ユーザがプログラムの中で特に重要と思う部分を選択し、TVISは、その部分だけを可視化する。

局所的な可視化を実現することにより、ステップ数が多いプログラムでも理解支援を実用的な状態で行えるようになると思われる。

参考文献

- 1) Roger S. Pressman (2001), Software Engineering A Practitioner's Approach 5th Edition, McGraw-Hill Science.
- 2) M. Sridharan, S. J. Fink, R. Bodik (2007), Thin Slicing, In Proc. the 2007 ACM SIGPLAN Conference on PLDI, pp.112-122.
- 3) Mark Weiser (1982), Programmers Use Slices When Debugging, Communications of the ACM Vol.25, pp. 446-452.
- 4) JavaCC Home Page, <http://javacc.java.net/>, accessed 2014-2-21.
- 5) H Agrawal, JR Horgan (1990), Dynamic Program Slicing, SIGPLAN Notices, Vol.25, No.6, pp.246-256.
- 6) C. Murphy et al (2006), How Are Java Software Developers Using the Eclipse IDE?, IEEE Software, Vol.23, No.4, pp.76-83.
- 7) Cheng Zhang et al (2013), Automated Breakpoint Generation for Debugging, JOURNAL OF SOFTWARE, Vol.8, No.3, pp.603-616.
- 8) Steven P. Reiss, Guy Eddon (2005), Visual Representations of Program Execution, DMS 2005, pp.315-320.
- 9) Jan Lönnberg et al (2011), Java Replay for Dependence-based Debugging, PADTAD '11, pp.15-25.
- 10) W. De Pauw et al (1998), Execution Patterns in Object-oriented Visualization, In Proc. 4th COOTS, pp. 219-234.