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

佐藤　拓弥
宮崎大学大学院工学研究科
sato@earth.cs.miyazaki-u.ac.jp

片山　徹郎
宮崎大学工学教育研究部
kat@cs.miyazaki-u.ac.jp

水久保　直哉
株式会社スカイコム
mizukubo@skycom.jp

田中　伸英
株式会社スカイコム
tanaka@skycom.jp

要旨

ソフトウェア開発工程において、デバッグは手間と時間のかかる工程である。このデバッグにかかる手間と時間の削減を目的として、我々の研究室ではデータ遷移可視化ツール *TFVIS* を開発した。*TFVIS* は、データ遷移可視化と実行フロー可視化によって、プログラム実行時の挙動把握を支援する。*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、プログラムが含む欠陥の特定を支援する。しかし、*TFVIS* は一部の制御構造や式にしか対応しておらず、有用性が高いとは言えない。そこで、例外処理を含む *Java* プログラムを対象とした適用範囲の拡大を行った。これにより、*TFVIS* の *Java* プログラム可視化ツールとしての有用性が向上したと言える。

1. はじめに

ソフトウェアの開発工程において、デバッグは手間と時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]。

効率よくプログラムの欠陥を特定するためには、プログラムの動的な挙動を理解することが重要である [3]。しかし、プログラムの動的な挙動は、一般的に不可視であるため、把握することが困難である [4]。

この問題を解決するため、我々の研究室では Java プログラムの動的な挙動を可視化するツール TFVIS を開発した [5]。

TFVIS は、データ遷移可視化と実行フロー可視化によって、プログラム実行時の挙動把握を支援する。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、プログラムが含む欠陥の特定を支援する。また、他の機能として、データ遷移を矢印を用いて示すことができる。これにより、プログラムの不具合から欠陥の特定を容易にする。しかし、TFVIS は一部の制御構造や式にしか対応しておらず、有用性が高いとは言えない。

そこで本稿では、未対応の制御構造の 1 つである、例外処理を含む Java プログラムを対象とした適用範囲の拡大を行う。具体的には、Try Catch 文を含むプログラムを可視化できるように拡張を行う。これにより、TFVIS の Java プログラム可視化ツールとしての有用性の向上を目指す。

2. TFVIS

今回適用範囲の拡大を行う TFVIS の機能と構造について、それぞれ以下で説明する。

2.1. 機能

図 1 に、TFVIS の外観を示す。TFVIS はソースコードからプログラムの構造を解析した構造情報と、実行時

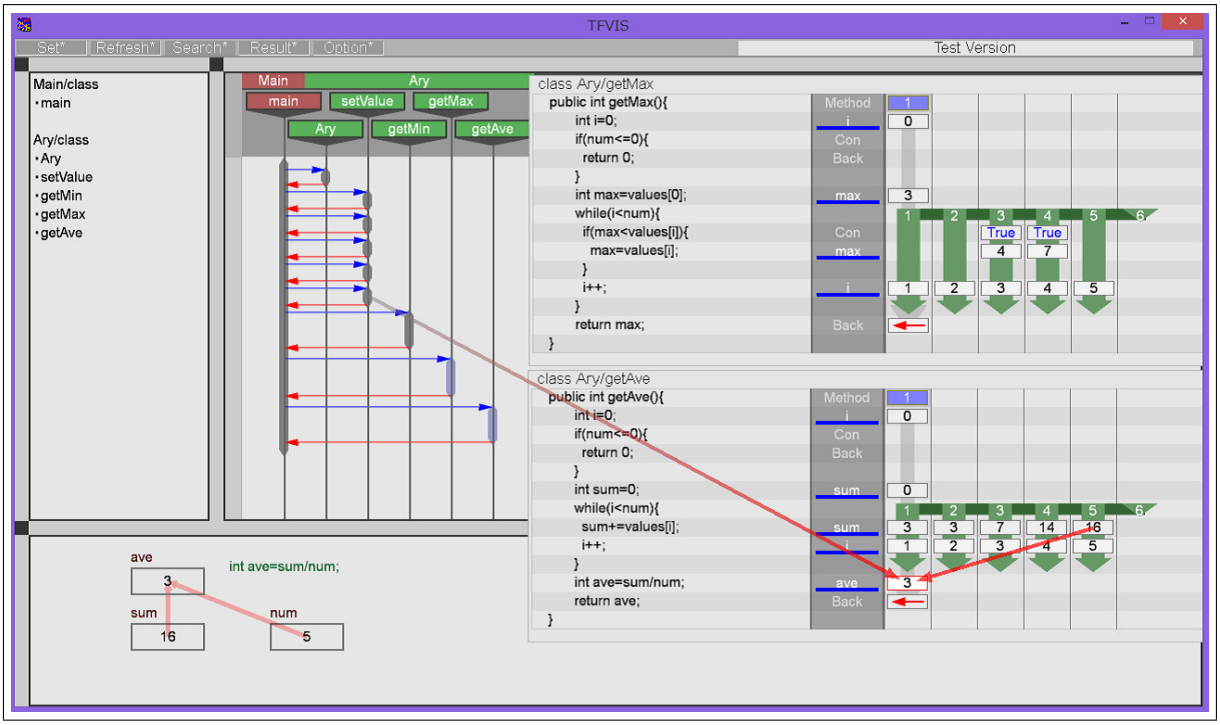

図 1. TFVIS の外観

の情報を基に、データ遷移図を生成する。データ遷移図は、ループ処理や分岐などによる処理の流れの変化、変数の更新によるデータの移り変わりなど、プログラム実行時の各メソッドの詳細な挙動を示す。

図 1 を例に説明する。図の右側に見える個々のウィンドウが、各メソッド内の処理を示すデータ遷移図である。ウィンドウの左側にメソッドのソースコードを、中央に各行で起こるイベントや変数名を、右側に処理の流れを示している。if 文や for 文によって、処理の流れが変化する場合、右側に濃い緑色で処理の流れの様子を描画する。また、メソッド中で変数の更新を行う処理がある場合、更新後の値をボックスに記述し、対応するソースコード右側に描画する。

ウィンドウの左側に見える縦に伸びるラインを描画してある図が、プログラム実行全体の処理の流れを示す実行フロー図である。ライン上にある太い部分がメソッドの活性区間であり、これを選択することで、該当するメソッドのデータ遷移図が確認できる。メソッドの活性区間から伸びる青い矢印はメソッド呼び出しを、赤い矢印はメソッドの処理の終了を示す。

TFVIS のもう 1 つの機能として、よる変数同士の関係を可視化するデータ遷移線がある。データ遷移線は、ある特定の変数が更新された際に、その更新に影響を与えた変数がプログラムのどの時点で生成されたものであるかを示す。例えば、「a=b+c」という式が存在するとき、「a」の値を選択することで、更新に関わった「b」と「c」の値が生成された時点を赤色の矢印を用いて示す。このデータ遷移線を活用することで、特定の変数がどのような変数同士の関係で生成されたかを確認できる。これにより、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。

さらに、TFVIS はプログラム全体の流れを、UML のシーケンス図 [6] を基にした実行フロー図によって可視化する。これにより、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。プログラム全体の処理を実行フロー図によって可視化することで、各メソッドの詳細な挙動を示すデータ遷移図の活用を補助する。

2.2. 構造

図 2 に、TFVIS の構造を示す。TFVIS は、解析部と可視化部から成る。また、解析部は、構造解析部、プロー

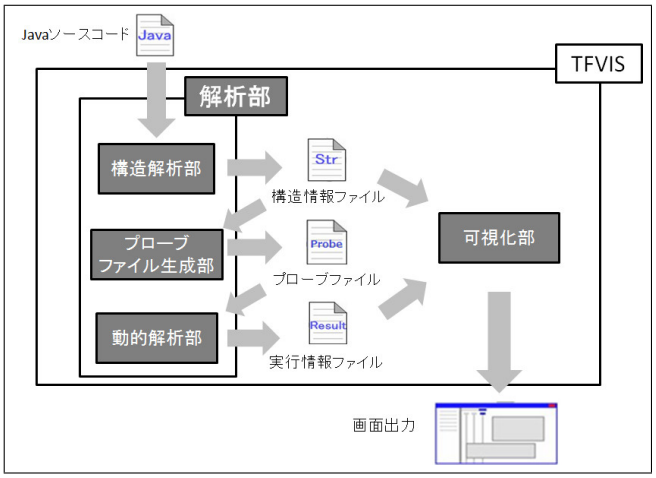

図 2. TFVIS の構造

ブファイル生成部、動的解析部から成る。

構造解析部では、プログラムの構造の解析を行い、解析結果を構造情報としてファイルに出力する。構造情報は、プローブファイル生成部でのプローブ挿入箇所の判断と、可視化部での図表の作成に用いる。構造解析部によって、ソースコードの各行で起こるイベントを取得する。イベントとは、可視化の基準となる特定の処理であり、各イベントはイベント種別の値を持つ。

プローブファイル生成部では、構造情報を基に、対象ソースコードにプローブを埋め込んだプローブファイルを生成する。プローブは、プログラム実行時の挙動の情報を出力する。なお、プローブにはいくつか種類があり、各コードで起きるイベントごとに挿入するプローブが変わる。

動的解析部は、プローブファイルから、実行時の挙動を解析し、解析結果を実行情報として出力する。具体的には、ソースコードにプローブを挿入したプローブファイルをコンパイルし実行することで、プローブが出力する実行情報を取得し、実行情報をファイルに出力する。

可視化部は、解析部が出力する構造情報と実行情報を基に、可視化を行う。

3. TFVIS の拡張

本章では、Try Catch 文への対応のために行った拡張について述べる。初めに、対応のために行った各部の拡張について述べる。次に、改良後の TFVIS のデータの流れについて述べる。

3.1. 各部の拡張点

Try Catch 文への対応のために行った拡張は、以下のとおりである。

- 構造解析部の新たなイベント種別の値の定義
- プローブファイル生成部の Try Catch 文に対する新たなプローブの定義
- プローブファイル生成部の Try Catch 文に対するプローブの挿入
- 可視化部の Try Catch 文のイベントに対する可視化

各拡張の詳細について、以下で述べる。

3.1.1 構造解析部の新たなイベント種別の値の定義

構造解析部において、Try Catch 文に対し出力するイベント種別の値を新たに定義する。

新たに定義したイベント種別の値は、Try ブロック開始の値 (380)、Try ブロック終了の値 (382)、Catch ブロック開始の値 (390)、Catch ブロック終了の値 (392) である。

3.1.2 プローブファイル生成部の Try Catch 文に対する新たなプローブの定義

プローブファイル生成部において、Try Catch 文のイベントに対して挿入するプローブを新たに定義する。

Try ブロックのイベント用のプローブを、Try 処理検出プローブとする。このプローブは、実行時のインスタンスの ID、メソッド ID、メソッド実行番号、行番号を引数とする。そして、可視化で用いる Try イベント ID(380)、インスタンス ID、メソッド ID、メソッド実行番号、行番号を、実行情報ファイルに出力する。

同様に、Catch ブロックのイベント用のプローブを、Catch 処理検出プローブとする。このプローブは、実行時のインスタンスの ID、メソッド ID、メソッド実行番号、行番号を引数とする。そして、可視化で用いる Catch イベント ID(390)、インスタンス ID、メソッド ID、メソッド実行番号、行番号を、実行情報ファイルに出力する。

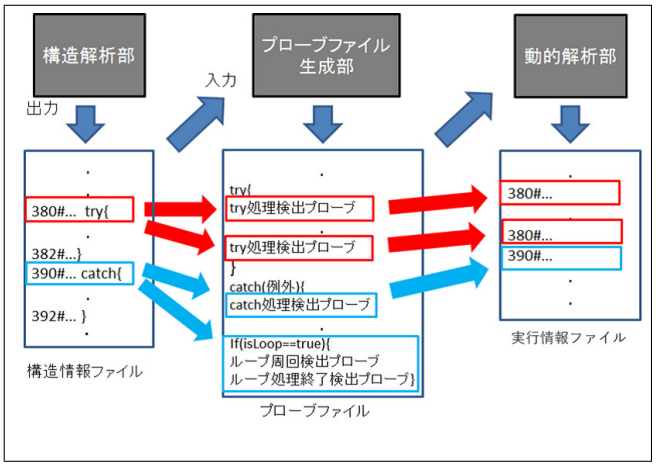

図 3. 解析部のデータの流れ

3.1.3 プローブファイル生成部の Try Catch 文に対するプローブの挿入

プローブファイル生成部において、Try Catch 文のイベントに応じて新たに定義したプローブを挿入する。

Try ブロック開始のイベントを読み込んだ場合、Try ブロック開始から Try ブロック終了までの全ての行の直前に、Try 処理検出プローブを挿入する。このときプローブが得る行番号の値は、直後の行の行番号である。また、Catch ブロック開始のイベントを読み込んだ場合、直後の行に Catch 処理検出プローブを挿入する。

ループ中に Catch ブロックを実行する場合、ループから抜ける処理が存在する。このような処理に対応するため、メソッド開始の行の直後に、ループ中かどうかを示す boolean 型の変 “isLoop”の定義を挿入する。そして、ループ中であれば、“isLoop” が true になるように変更を行った。また、Catch ブロック終了の行の直前に、”isLoop” が true であれば、既存のループ周回検出プローブとループ処理終了検出プローブを実行する if 文を挿入する。

3.1.4 可視化部の Try Catch 文のイベントに対する可視化

例外処理のイベントの発生に対し、データ遷移図の直前の実行の箇所に赤色で “Catch”と記述したボックスを配置する。

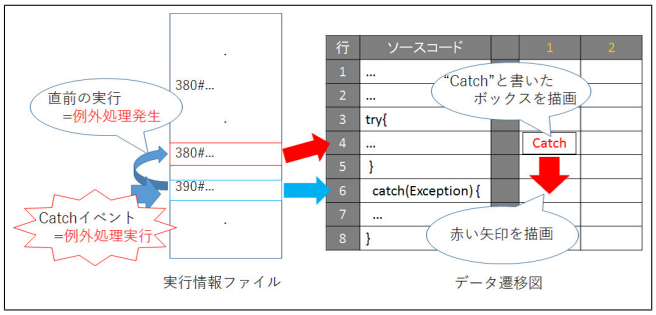

図 4. 可視化の流れ

また、例外処理の発生箇所と実行箇所を結ぶ赤色の矢印を記述する。

3.2. 改良後のデータの流れ

拡張後の TFVIS の解析部と可視化部における、詳細なデータの流れについて、以下で述べる。

3.2.1 解析部のデータの流れ

図 3 に、拡張後の TFVIS に、Try Catch 文を含むプログラムを適用した際のデータの流れを示す。

初めに、構造解析部の拡張によって、Try Catch 文についてのイベントを取得する。図 3 から、構造情報ファイルが Try ブロック開始の値 (380)、Try ブロック終了の値 (382)、Catch ブロック開始の値 (390)、Catch ブロック終了の値 (392) を持っていることが分かる。

次に、構造解析部で取得した Try Catch 文についてのイベントによって、プローブファイル生成部が新たに定義したプローブを挿入する。図 3 から、Try ブロックの全ての行の直前に Try 処理検出プローブを、Catch ブロック開始の行の直後に Catch 処理検出プローブを挿入していることが分かる。また、ループ中であれば、ループ周回検出プローブとループ処理終了検出プローブを実行する if 文を挿入していることもわかる。

最後に、プローブを埋め込んだプローブファイルを動的解析部で実行することで、実行情報を取得する。

3.2.2 可視化部の流れ

図 4 に、拡張後の TFVIS に、Try Catch 文を含むプログラムを適用した際の可視化の流れを示す。

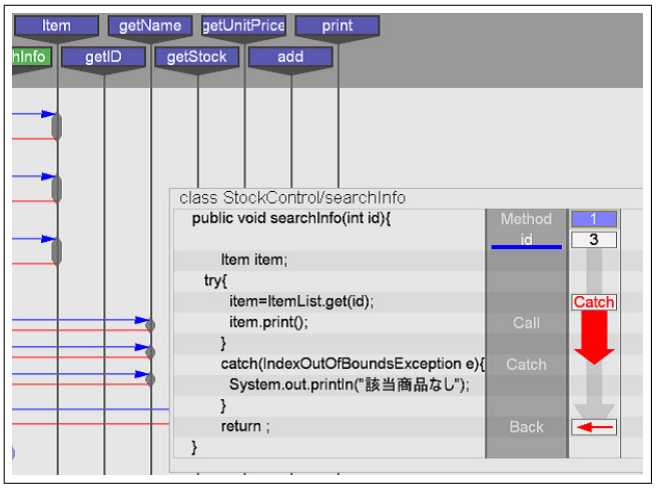

図 5. SearchInfo メソッドの可視化

構造情報に基づき、実行フロー図とデータ遷移図上のソースコードを描画する。また、実行情報に基づき、データ遷移図を描画する。

可視化部が Catch イベント ID を含む実行情報を読み込んだ場合、可視化部は Catch イベントの行を例外処理実行箇所として、その直前の実行の行を例外処理発生箇所として保持する。そして、例外処理実行箇所に赤色で“Catch”と記述したボックスを、例外処理発生箇所から例外処理実行箇所まで赤色の矢印を描画する。

4. 適用例

本章では、3 章で説明した TFVIS の拡張によって、Try Catch 文を含むプログラムが、正しく可視化できることを確認する。適用例として、Java で記述した「在庫管理プログラム」を適用し、例外処理を正しく可視化することを示す。また、「与えられた数の最小、最大、平均値を計算するプログラム」を適用し、TryCatch 文を含むプログラムの可視化において、データ遷移線が正しく機能することを示す。

4.1. 在庫管理プログラム

図 5 に、「在庫管理プログラム」において、Try Catch 文を含む SearchInfo メソッドを可視化したデータ遷移図を示す。このメソッドは、「ID」を入力として、その「ID」を持つ在庫の「ID」と「名前」、「在庫数」、「単価」を表示するメソッドである。

今回作成した在庫は 3 つのみのため、3 つの在庫のインスタンスを生成し、リストに格納した。このときユーザが「ID」を 3 と入力したと仮定すると、リストの 4 番目を参照したことになり、「IndexOutOfBoundsException」という例外が発生する。図 6 から、リストを参照する行に“Catch”のボックスと、このボックスから、発生した例外処理の行までの赤色の矢印を表示しており、データ遷移図で例外処理の発生を正しく可視化していることが分かる。

4.2. 最大、最小、平均値を計算するプログラム

図 6 に、「与えられた数の最小、最大、平均値を計算するプログラム」において、TryCatch 文を含む getAve メソッドを可視化したデータ遷移図を示す。このメソッドは、与えられた数の平均値を計算するメソッドである。

今回の適用例では、このメソッドに故意に例外が発生する処理を記述し、例外処理内の処理でデータ遷移線が機能するか確認を行った。今回基点として選択した更新値は、getAve メソッド内のループ処理の 5 回目のループで更新された変数「sum」の「16」という値である。この「sum」は、「sum+=values[i]」という式からから計算される値である。この更新値の算出に用いた「sum」、「values[i]」、「i」を生成した箇所をそれぞれ赤色の矢印で描画している。このことから、例外処理を含むプログラムにおいても正しくデータ遷移線が機能していることが分かる。

5. 考察

本稿では、未対応の制御構造の 1 つである、例外処理を含むプログラムへの適用を目的とした拡張を行った。具体的には、Try Catch 文を含むプログラムを可視化できるように拡張を行った。本章では、初めに TFVIS の拡張の評価を述べる。次に、関連研究について述べる。最後に、TFVIS の課題について述べる。

5.1. 評価

既存の TFVIS では、Try Catch 文を含むプログラムを適用した場合、コンパイルエラーが発生し、可視化を行うことができなかった。

Try Catch 文を含むプログラムを可視化するため、初めに、構造解析部において、Try Catch 文に対して新た

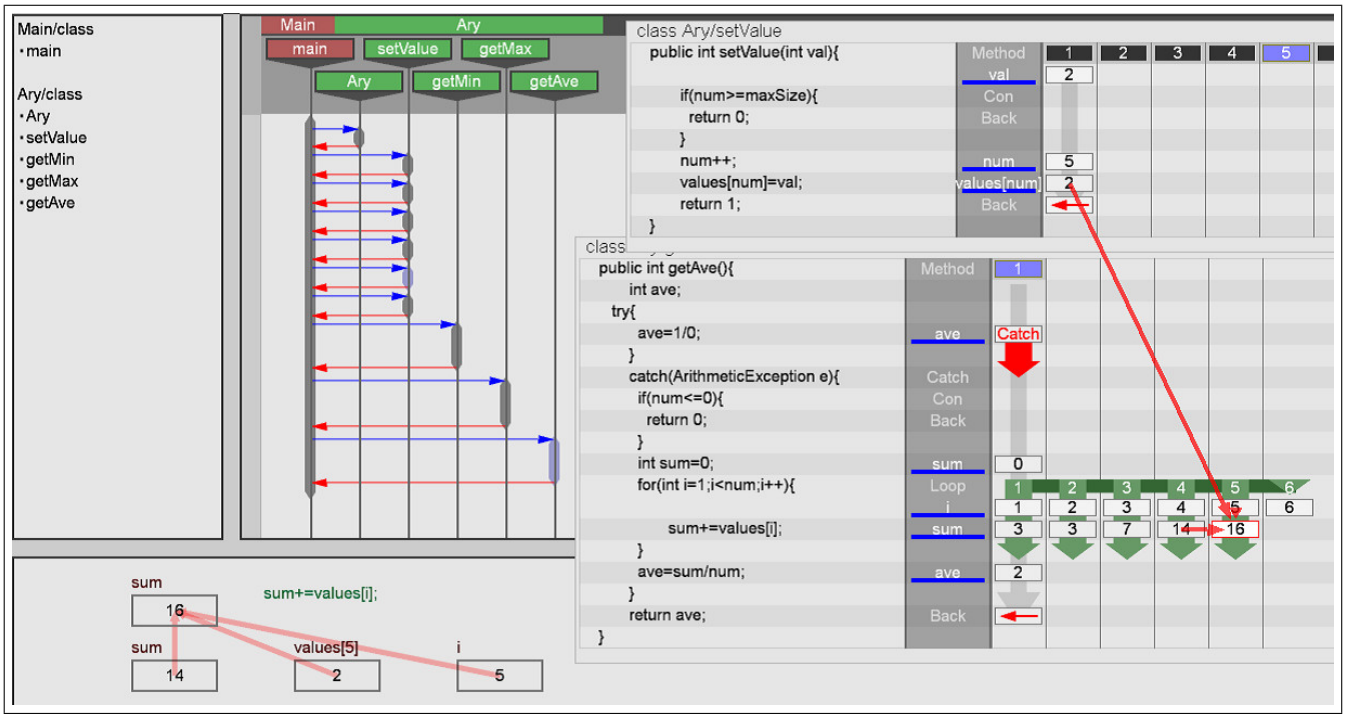

図 6. getAve メソッドの可視化

に定義したイベント種別の値を用いることでイベントを取得する拡張を行った。

次に、プローブファイル生成部において、Try Catch 文についてのイベントに対して、新たに定義した Try Catch 文の実行時の情報を出力するプローブを挿入する拡張を行った。

最後に、可視化部において、Catch イベントに対して、直前の実行に赤色で “Catch” と記述したボックスを描画し、例外処理の発生箇所と実行箇所を結ぶ赤色の矢印を描画する拡張を行った。

以上の拡張から、既存の TFVIS では可視化できなかった Try Catch 文を含むプログラムを、拡張後の TFVIS は可視化できる。また、Try Catch 文を含むプログラムで、既存のデータ遷移線が機能することも確認できた。このことから、Try Catch 文への対応により、TFVIS の実用性が向上したと言える。

5.2. 関連研究

以下に、TFVIS と関連研究との比較を述べる。

- ブレークポイントデバッグ

ブレークポイントは、最も多用されているデバッグ支援手法の 1 つである [7]。ブレークポイントを用いたデバッグでは、プログラムの実行を任意の箇所で停止し、停止した時点での各変数の値など、プログラムの実行状況を確認することができる。

ブレークポイントデバッグには、ブレークポイントを設置する箇所の選定が難しいという問題点が存在する。プログラムの欠陥を特定するのに適当な設置箇所を選定するためには、ユーザの知識と経験が必要である [7]。

これに対して TFVIS は、ユーザが欲する情報を保持するメソッドを選択するだけで必要な情報を得ることができ、ユーザの能力に依存せずに使用できるという点で優れていると言える。

さらに、TFVIS による可視化では、メソッドやプログラム全体の流れを俯瞰することができる。また、データ遷移線を活用することで、変数同士の依存関係を把握することができる。これらの機能から、ブレークポイントによるデバッグに比べ、ある変数がどのような経緯で作られたのか調べることができる、という点で優っていると言える。

- JIVE

  JIVE[8] (Java Interactive Visualization Environment) は、Java プログラムの実行を可視化するツールである。

  JIVE は、実行時の処理から UML のオブジェクト図とシーケンス図を生成する機能を持つ。また、クエリーによる問い合わせに対応しており、例えば、「メソッド “func” が返り値に NULL を返すのはどこか」といった問い合わせが可能である。問い合わせで発見した処理は、シーケンス図上でハイライトされ、プログラム実行時の挙動把握を支援する。

  JIVE と TFVIS を比較した場合、JIVE には、データ遷移のような変数同士の依存関係を示す機能はない。変数更新の問い合わせが可能であるが、データ遷移のような依存関係を調べる場合には、繰り返し問い合わせを行う必要がある。そのため、不審な値を見つけた際に、その原因を探るといった作業には、TFVIS がより効果的であると言える。

- Code Canvas

  Code Canvas[9] は、IDE である Visual Studio のズーム可能なサーフェスである。Code Canvas には、スタックトレースから例外発生箇所とメソッド呼び出し関係を、複数のソースコードにわたって赤い矢印で描画する機能を持つ。

  TFVIS と比較した場合、Code Canvas は例外処理の呼び出し関係の把握をスタックトレースを可視化することで支援するが、例外発生箇所の変数の値や処理の流れなどは調査できない。TFVIS による可視化では、例外発生箇所に不審な値がある可能性が高いため、そこからデータ遷移を辿っていくことで効率よく欠陥を特定できる。この点で、TFVIS による例外処理の可視化は有用であると言える。

5.3. 今後の課題

以下に、TFVIS の課題について述べる。

- レスポンスの遅さ

  TFVIS が可視化を行うためには、読み込む対象のプログラムが終了するまで待たなければならない。また、可視化を行うためには、いくつかの手順を踏まなければならず、1 回の可視化を行うのに、2 分程度の時間がかかってしまう。

- 入力待ち状態の発生を含むプログラムへの対応

  TFVIS はユーザからの入力を受け取る機能を持たない。そのため、入力待ち状態の発生を含むプログラムを適用した場合、解析部の実行が終了せず、実行情報を得ることができない。

- 問い合わせ機能の実装

  TFVIS は問い合わせ機能を持たない。この問題は可視化対象のプログラムの処理が増加するほど、大きな手間となる。そのため、問い合わせ機能を実装する必要があると考えている。

- マルチスレッドプログラムへの対応

  TFVIS はマルチスレッドプログラムを可視化できない。マルチスレッドプログラムでは、スレッド間でデータのやり取りが行われるため、処理が複雑になりやすい。もし、マルチスレッドによる複雑な処理を可視化できれば、TFVIS の有用性がより高まると言える。

以上の課題のうち、レスポンスが遅いという問題点について詳しく述べる。上記で述べたように、TFVIS が可視化を行うためには、読み込む対象のプログラムが終了するまで待たなければならない。また、可視化を行うためには、いくつかの手順を踏まなければならず、1 回の可視化を行うのに、コードの大きさに関わらず 2 分の時間がかかってしまう。これは、プログラムを任意の点で停止し、実行状況を把握できるブレークポイントに比べ、レスポンスが遅いと言える。

しかし、関連研究でも述べたように、TFVIS による可視化には、メソッドやプログラム全体の流れを俯瞰できる、データ遷移線を活用することで変数同士の依存関係を把握できるといった強みが存在する。

そこで、ブレークポイントデバッグを支援するツールとして TFVIS の改良を行う。具体的には、Java 言語の IDE として広く使われており、プラグインの開発が可能な Rclipse[10] のブレークポイント機能と連携を行う。ブレークポイントで停止した時点で、その時点までの実行フロー図やデータ遷移図を TFVIS で表示することで、TFVIS のレスポンスの問題とブレークポイントデバッグの処理の流れの把握が困難であるという問題点を解決

できるのではないかと考える。さらに、ブレークポイント機能と合わせて、データ遷移線による変数同士の依存関係を確認することで、より効率的な欠陥の特定が可能になると考える。

以上の点から、TFVIS の今後の展望として、eclipse プラグインとして改良を行い、ブレークポイント機能と連携することで、ブレークポイントデバッグを支援するツールを目指す。

6. おわりに

本稿では、未対応の制御構造の 1 つである、例外処理を含む Java プログラムを対象とした適用範囲の拡大を行った。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。また、データ遷移線の機能から、データ遷移を辿ることができ、不具合から欠陥の特定を支援することができる。

しかし、既存の TFVIS では可視化できないプログラムが存在する。Java プログラムが持つ基本的な構文を含むプログラムが可視化できないことは有用性に欠けることを意味している。

そこで本稿では、Try Catch 文を含む Java プログラムを対象とした、TFVIS の適用範囲の拡大を行った。今回の拡張により、TFVIS の Java プログラム可視化ツールとしての有用性が向上したと言える。

今後の課題を以下に示す。

- レスポンスの遅さ
- 入力待ち状態の発生を含むプログラムへの対応
- 問い合わせ機能の実装
- マルチスレッドプログラムへの対応

参考文献

[1] Thomas D. LaToza, Gina Venolia, and Robert DeLine: Maintaining mental models: a study of developer work habits, Proceedings of the 28th international conference on Software engineering, pp.492-501 (2006).

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

[3] Jonathan Sillito, Gail C. Murphy, and Kris De Volder: Asking and Answering Questions During a Programming Change Task, IEEE Transactions on Software Engineering, Vol.34, No.4, pp.434-451 (2008).

[4] Andreas Zeller, (訳: 中田秀基, 今田昌宏, 大岩尚宏, 竹田香苗, 宮原久美子, 宗形紗織): 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).

[5] Hiroto Nakamura, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, Kentaro Aburada and Naonobu Okazaki: 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).

[6] Dan Pilone, Neil Pitman, (訳: 原 隆文): UML2.0 クイックリファレンス, オライリー・ジャパン (2006).

[7] Cheng Zhang, Juyuan Yang, Dacong Yan, Shengqian Yang and Yuting Chen: Automated Breakpoint Generation for Debugging, Journal of Software, Vol.8, No.3, pp.603-616 (2013).

[8] Demian Lessa, Bharat Jayaraman and Czyz Jeffrey: JIVE: A Pedagogic Tool for Visualizing the Execution of Java Programs. Technical Report 2010-13, Department of Computer Science and Engineering, University at Buffalo (2010).

[9] R. DeLine and K. Rowan: Code Canvas: Zooming towards better development environments, in Proc. of the 32nd International Conference on Software Engineering, Vol.2, pp.207-210 (2010).

[10] Eclipse Foundation: Eclipse - The Eclipse Foundation open source community website., `https://eclipse.org/`