Vol.2013-SE-182 No.4
2013/10/24

フレームワークアプリケーションに於ける副作用の兆候を抽出する動的解析手法

久米 出[1,a)]　新田 直也[3,b)]　中村 匡秀[2,c)]　柴山 悦哉[4,d)]

概要：プログラムのデバッグには検出された不具合からその原因を特定するために、多くの時間や労力を費して制御やデータの流れを辿る作業が必要である。多くのデバッグ支援手法は、作業者によるデバッグ対象プログラムの実装に関する知識を想定している。しかしながらフレームワークアプリケーションのように、既知のコードと未知のコード同士が相互に呼び出し合うようなプログラムの場合にはこうした想定が成り立たない。
本論文ではフレームワークアプリケーションに於ける副作用に焦点を当てたプログラム理解とデバッグ支援の手法を提案する。我々の手法は副作用と共に発生する挙動を抽象化した**兆候**をプログラムトレースから抽出する点に新規性を有する。兆候は作業者が副作用の有無を判断する根拠として利用される。またオブジェクトやメソッド呼び出しと関連付ける事によって副作用の発生過程の理解を支援する事も期待されている。本提案手法を実用的なアプリケーション例題に対して適用し、その結果を評価する。

1. Introduction

フレームワークアプリケーションは設計と実装の再利用が可能なフレームワーク上に構築されたアプリケーションである。フレームワークアプリケーションはフレームワークとアプリケーション固有部分に分割される。アプリケーション固有部分のコードはフレームワークを拡張する形でその固有な機能を実装する [1]。近年では多くのアプリケーションが何らかのフレームワーク上に構築されている。ソフトウェア開発の現場ではフレームワークアプリケーションをデバッグする機会も増えている。そのデバッグ支援手法を研究する際には、フレームワークアプリケーション特有の構成や実行時の挙動を考慮する必要がある。

通常、アプリケーション開発者はフレームワークの実装上の詳細は知悉しておらず、その正しい利用の効率的な学習がフレームワーク利用に於ける大きな課題となっている [2], [3] 。フレームワークアプリケーションの不具合の主要な原因の一つとしてアプリケーション開発者によるフレームワークの誤利用が挙げられる [4]。実行時にはフレームワークがアプリケーション固有なコードを呼び出す所謂**制御の反転** *(inversion of control)*[5] や**制御の再反転** *(re-inversion of control)*[6] と呼ばれる複雑な呼び出し関係が形成される。

一般的なデバッグ作業では、バグの所在を特定するために、デバッガを用いて変数の値やオブジェクトの状態を調査しながら、エラーの発生箇所から制御やデータの流れを逆に辿る作業に多くの労力が費される [7]。この作業に必要な労力の軽減を目的として様々なデバッグ支援手法が提案されている [8], [9], [10], [11], [12], [13] 。こうした既存の手法は作業者がデバッグ対象となるプログラムの実装の詳細に知悉している事を想定している。

しかしながら、フレームワークアプリケーションのデバッグに関してはこの前提が成立しない事が多い。先に述べたように、フレームワークアプリケーションの実行時にはしばしば制御の反転と再反転入れ子上に実行される [6]。よってそのデバッグ作業に於いては、本来作業者が直接触れる事のないフレームワークの実装を調査する事になる。フレームワークを誤利用しているアプリケーション固有なコード (**逸脱コード** *(deviant code)*) の修正に関しても同様な問題点が指摘されている [4]。

本論文では、フレームワークアプリケーションの不具合のうち、特に副作用に焦点を当てたデバッグ支援手法を提

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[2] 神戸大学
Kobe University
[3] 甲南大学
Konan University
[4] 東京大学
The University of Tokyo
[a)] kume@is.naist.jp
[b)] n-nitta@konan-u.ac.jp
[c)] masa-n@cs.kobe-u.ac.jp
[d)] etsuya@ecc.u-tokyo.ac.jp

案する。本手法は副作用の発生に伴う実行時の挙動に関する特徴を抽出し、デバッグ作業に利用する点に新規性を有する。これらの抽出された特徴は**兆候** *(symptom)* と命名される。兆候は副作用を示唆する “bad smell” であり、副作用の有無の判断に利用される。また、核心クラス (core class)[1] の役割や API 等、アプリケーション開発者に公開された情報のみを利用して、副作用の生成過程に関するより詳細な理解を支援する事が期待される。

我々は実行時に例外が発生したプログラムトレースから兆候を抽出し、データや制御の依存性に基づいて関連付け、関連付けた結果を可視化する動的試作ツールを試作した。これに加えて本試作ツールは兆候を実装するソースコードの実行履歴も合わせて出力する機能を有している。試作ツールを用いて第三者が開発した実用的なフレームワークアプリケーションの不具合事例を解析し、本提案手法を評価する。

本論文の以降の部分は以下の通り構成されている。まず第 2 節で議論の土台となる基本概念を説明する。第 3 節で具体的な事例を用いてフレームワークアプリケーションのデバッグの問題点を指摘する。第 4 節で提案手法を説明し、具体的な不具合事例に手法を提案した結果を第 5 節で評価する。第 6 節で関連研究に関して述べた後に第 7 節で結論を述べる。

2. 基本概念

2.1 オブジェクトと永続変数

我々の提案手法は Java 言語で記述されたフレームワークとそのアプリケーションを対象としている。Java のオブジェクトとしてクラスインスタンスと配列が扱われる。本稿ではクラスインスタンスのインスタンス変数や、配列の配列要素を**永続変数**と呼ぶ事にする。この名称はこれらがメソッドの実行後にも値を保持し続けるという性質に由来する [14]。

Java 言語のオブジェクトは `instanceof` 演算子等でそれ自身が値として利用される場合がある。値として利用されているオブジェクトを**値オブジェクト**と呼ぶ事にする。オブジェクトの別の役割として、永続変数に値を保持する事にってその値の参照を可能にする事が挙げられる。このように用いられる場合、そのオブジェクトをその値の**運び手**と呼ぶ事にする。代入される値が数値或いは値オブジェクトであるような運び手を**値保持者**と呼ぶ事にする。

実行時にある命令文 (statement) 中で参照されているオブジェクト **obj** に関して、そのオブジェクトが参照されるまでの履歴に直接間接的に関与した運び手を特定出来る。これは以下のように **obj の参照経路**を取得する事によって可能になる。。起点となる **obj** の参照からデータフローを逆に辿り、最初に遭遇した永続 変数参照の運び手に対して同様の処理を再帰的に実施する事によってその参照経路が取得出来る。

2.2 アプリケーションフレームワーク

第 1 節で述べたように、フレームワークアプリケーションはフレームワークとアプリケーション固有部分に分割される。フレームワークに属するクラスは**フレームワーククラス**、そのメソッドを**フレームワークメソッド**と呼ぶ事にする。アプリケーション固有部分のクラスとそのメソッドは**アプリケーショ ン固有クラス**、**アプリケーション固有メソッド**と呼ばれる。これらのクラスからそれぞれ、Java のライブラリに所属する Java ライブラリクラスが参照されている。

本論文中でそのクラスやメソッドの所属を示すためにクラス名の前に記号が付与される。フレームワーククラスやメソッドに対してはアルファベット **F** が、アプリケーション固有クラスやメソッドに対しては **A** が、Java ライブラリクラスに対しては **J** が付与される。

フレームワークにはアプリケーション固有の機能を実現するために、幾つかのクラスが**ホットスポット**として指定されている。アプリケーション開発者はホットスポットを拡張する形でアプリケーション固有の機能を実現する [15]。フレームワークアプリケーションの実行時には、所謂**制御の反転** [5] の過程の一部として、フレームワークからホットスポットを拡張したアプリケーション固有メソッドが呼び出される。これとは逆にアプリケーション固有メソッドからのフレームワークメソッドの呼び出しもしばしば発生する。このような呼び出しは制御の反転の準備や、GUI の初期化等の一部として実行される。

このように、フレームワークアプリケーションでは、フレームワークとアプリケーション固有部分の間には相互呼び出し関係が、時には入れ子状に形成されている [6]。これらに加えて Java ライブラリクラスのメソッドはフレームワークメソッドかアプリケーション固有メソッドから呼び出される。我々はフレームワークアプリケーションに於ける命令文の実行に対して**実行モード**を以下のように定義する。命令文がフレームワークメソッド内で実行されているか、或いはフレームワークメソッドから呼び出された Java ライブラリクラスのメソッド内で実行されている場合、その命令文は**フレームワークモード**で実行されていると言う。アプリケーション固有メソッドに対しても同様に**アプリケーション固有モード**が定義出来る。Java ライブラリクラスのメソッドがフレームワークモードで呼び出されている事を示すためにクラスやメソッド名に対して **FLib** という記号が付与される。アプリケーション固有モードの場合には **ALib** が付与される。

3. 例題

3.1 フレームワークアプリケーション例

本論文では第三者によって開発された実用的なフレームワークのアプリケーションを例題として使用する。本フレームワークアプリケーションは簡単な機能を有する UML 編集プログラム [*1] であり、一般的なグラフ編集アプリケーション用のフレームワーク [*2] 上に構築されている。

フレームワーククラスにはグラフの節点や辺を含む一般的なグラフの要素を表現するものが含まれる。フレームワークはこれらのグラフ要素の生成、削除、移動に伴う内部状態の更新や表示を担当する。アプリケーション固有クラスには UML クラスや多項 UML 関連を含む UML モデル要素を実装するものが含まれている。N-項 UML 関連は一つの節点とそれから延びる N 本の辺から構成される。前者はフレームワークの節点クラスを、後者はフレームワークの辺クラスを拡張する形で実装されている。

利用者による図式中の UML 要素の操作に対応して AWT/Swing イベントが作成され、フレームワーク内のイベントハンドラーに渡される。フレームワークはイベントを処理する過程でホットスポットを実装するアプリケーション固有メソッドを呼び出す。アプリケーション固有部分から呼び出されるフレームワークメソッドには、例えば UML モデル要素の表示内容を変更するものが含まれている。このように本例題アプリケーションではユーザによる操作を処理する度に、フレームワークとアプリケーション固有部分の間で入れ子状の相互呼び出しがしばしば発生している。

3.2 デバッグに於ける問題点

本 UML 編集プログラムは 3-項 UML 関連の節点の削除を試みると例外が発生するという不具合を有している。図 1 に例外の発生事例を示す。エラーメッセージには例外発生に至るメソッド呼び出し鎖が表示されている。これによって (1)Java ライブラリクラス `ArrayList` オブジェクトに対する `get()` メソッド呼び出しで例外が発生している事、かつ (2) このリストオブジェクトがこの時点で空になっていた事が示されている。

メソッド `get()` はフレームワークメソッド `FigNode.dispose()` 内から呼び出されている。フレームワークメソッドはさらに `ModelNodeFig.dispose()` から呼び出されている [*3]。このメソッドはこの呼び出し鎖内に於ける唯一のアプリケーション固有メソッドである。よってこの呼び出しの前後で実行モードが二度切り替わっている。

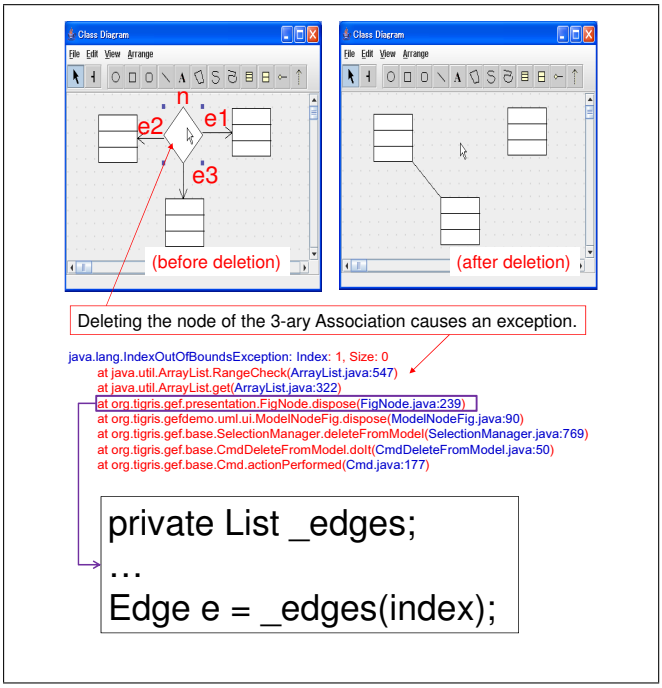


図 1 節点削除に伴う例外発生

デバッグ作業を `ArrayList.get()` 呼び出しから開始する場合、このメソッド呼び出しに関してどこに誤りが有ったのかを判断する必要が生じる。この呼び出しの受け手であるリストオブジェクトに関しては、この呼び出しの時点で空になっている事が問題なのか、言い換えればこのリストの状態に関して副作用が発生していたのか否かの判断が必要となる。この疑問に対する答えは図 1 のメソッド呼び出し鎖からは得られないため、デバッガを用いてこのリストオブジェクトが空になった過去の計算過程を調査する事になる。

こうした調査はこのリストオブジェクトに対して `get()` メソッドを直接呼び出している `FigNode.dispose()` だけに留まらないかもしれない。呼び出し鎖をさらに上に辿るだけでなく、例外発生時には Java 仮想機械のスタックから除かれていた呼び出しの調査まで必要となるかもしれない。

さらに、このメソッドの呼び出し鎖にはアプリケーション固有メソッドは `ModelNodeFig.dispose()` ただ一つしか含まれていない事に留意すべきである。他のメソッドは全てフレームワークメソッドか Java ライブラリクラスのメソッドであり、作業者にとって通常その実装は未知である。。このように実装の詳細が明らかでないフレームワーク内部であっても副作用が発生するか否かを効率的に判定するための新しい手法が求められている。そのような手法が存在するのであれば、このリストオブジェクトの現在の状態が副作用によるものなのか否かのみを判定するためだけに未知のコードと格闘する必要が無くなるのである。

[*1] http://gefdemo.tigris.org/ : version 0.10.5b
[*2] http://gef.tigris.org/ : version 0.10.5a
[*3] クラスの所属はパッケージ名で判別される。フレームワーククラスは `org.tigris.gef` で始まるパッケージ名を持ち、アプリケーション固有クラスは `org.tigris.gefdemo` で始まるパッケージ名を持つ。

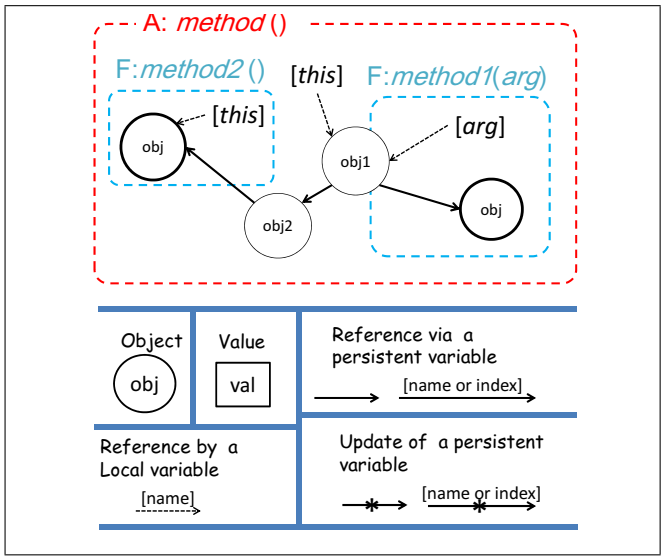


図 2 Aliasing 兆候

4. 提案手法

4.1 着想

フレームワークアプリケーションに特有なデバッグ上の問題を解決するために、副作用の発生を示唆する**兆候** *(symptom)* を抽出し、実際に副作用が発生しているか否かの判断に利用する手法を提唱する。兆候は過去の計算過程で発生した “bad smell” であり、副作用の発生に付随する挙動を表現している。本提案手法では三種類の兆候をプログラムトレースから抽出し、依存性解析を利用して例外発生等のエラーと関連付ける。兆候に対してはさらにオブジェクトやメソッド呼出しが関連付けられており、こうした情報はより詳細な調査に利用される。

4.2 兆候

兆候はプログラムで実行される命令文で参照されるオブジェクトとその状態の操作を包含するように定義されている。こうした操作には (1) そのオブジェクトへの参照経路、(2) 状態更新命令、(3) 更新された状態を用いた値の計算や条件分岐、が含まれる。

Aliasing 兆候はオブジェクトに対する**別名** *(alias)*、即ち複数の異なる参照経路の存在を示すものである。フレームワークアプリケーションの実装に於いては Aliasing が不可避であるために、この種の兆候は必ずしも副作用の存在を意味するわけではない。しかしながら、別名はプログラムの理解を困難にし [14], [16]、かつ副作用に伴うものと考えられてきた [17]。

Aliasing はオブジェクト指向フレームワークアプリケーションの実行時にあまりにも多数発生するため、本手法では値オブジェクトや値保持者として利用されるオブジェクトへの別名に限定して解析を実施する。これによって値の計算や制御に直接関与するオブジェクトへの参照経路のみを取り扱う事になる。

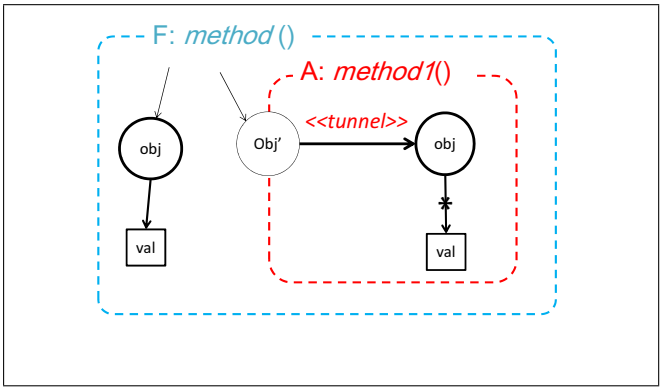


図 3 Hidden Update 兆候

図 2 では実行モードが切り替わる際のオブジェクト **obj** への別名の存在が示されている。このオブジェクトに対してはオブジェクト **obj1** と **obj2** の永続変数を経由した参照経路が存在し、フレームワークメソッド **method1** と **method2** 内ではこれらの参照経路経由で **obj** が参照されている。この図では永続変数の名前は省略されている。

Hidden Update 兆候はあるオブジェクトに対して実行モードを跨いだ (1) 参照経路の形成、(2) 状態変更、(3) 変更された状態の利用、の発生を示す。Hidden Update 兆候は実行時にある実行モードの視界外で発生した「予期せぬ効果 (unexpected result)」[17] を特定するために導入された。我々はこの兆候はフレームワークとアプリケーション固有部分の間の挙動に関する不整合や競合 [6] の発見にも有用であると考えている。

Hidden Update 兆候は以下の条件の元で成立する。まず、あるメソッド呼び出しによって実行モードが変更されている。次に呼び出されたメソッド内部で、パラメータからあるオブジェクトが取得され、その状態が変更される。ここで該当パラメータは取得されたオブジェクトとは別のオブジェクトでなければならない。則ち、そのパラメータから状態が変更されたオブジェクトに至る参照経路には一つ以上の永続変数参照が含まれている。最後に更新されたメソッドの状態はメソッドの呼び出し元で利用される。このようなオブジェクトを**隠れ更新対象**と呼ぶ事にする。

Hidden Update 兆候の事例を図 3 に示す。フレームワークメソッド `method()` がアプリケーション固有メソッド `method()` から呼び出されている。オブジェクト **obj** はアプリケーション固有メソッド `method1()` の内部でパラメータ **obj'** 経由で取得され、状態が変更されている。変更された状態は呼び出し元であるメソッド `method()` で参照されている。

Obj はアプリケーション固有部分の内部で取得され、その状態が変更されている。これらの操作はこのフレームワークメソッド (`method()`) から隠蔽されている。これらの操作は、もしフレームワーク側がそれを予期していない

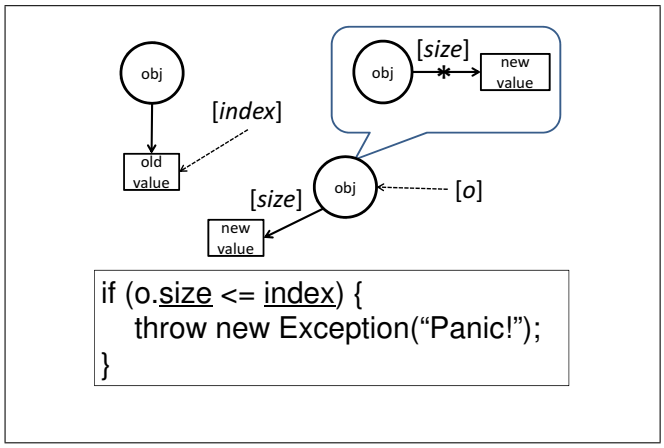


図 4 Outdated State 兆候

のであれば、副作用と見做されるべきである。

Outdated State 兆候はオブジェクトの期限切れとなった (outdated) 状態と更新された状態を合わせた利用を示す。こうした利用の問題点を考えるために、空でないリストがある時点で空になった場合を考えよう。仮にプログラムが、リストが空にされる以前にその長さを取得しているものとする。この取得されたリスト長はリストが空になった時点で既に期限切れとなっているが、プログラムがこの古いリスト長を信頼してリストから要素を取り出そうとすればエラーが発生するであろう。(図 4 を参照)

期限切れと判定された値の利用は必ずしも不具合を意味しない。実際にオブジェクトの古い状態と新しい状態の比較が必要とされる事は珍しくない。しかしながら、もしある状態変更が副作用を意味する場合、更新された状態と期限切れとなった状態を合わせた利用はバグとなる事が多いと思われる。またこうしたバグを解消するためには、その現象が明示化される事が望ましいであろう。以上が我々が Outdated State 兆候を導入した動機である。

4.3 実装

我々はフレームワークアプリケーション内で発生した例外と関連する兆候を特定し、可視化する試作ツールを開発した。図 1 に示された例外発生事例に関連する兆候を可視化した結果を図 5 と図 7 に示す。可視化処理は Swing の HTML API を利用して実装されている。各兆候はその種類とそれぞれに一意的な ID 番号によって表示されている。例えば `Aliasing#1784` は 1784 番の ID 番号を持つ Aliasing 兆候を表している。

第 3 節の例外はフレームワークモードで実行される Java library クラスのメソッド `ArrayList.get()` の内部で発生している。本試作ツールはこの例外発生元であるメソッド呼び出しに関して 3 通りの依存性解析を実施し、それぞれに関連する解析結果を表示に反映させる。

一つ目の依存性解析はメソッド `ArrayList.get()` の呼び出しを決定した制御に関するものである。二つ目の依存性解析はメソッド呼び出しのそれぞれのパラメータに関して実施される。三つめの依存性解析は呼び出されたメソッド内部で例外を発生させた `throw` 命令の制御に関して実施される。

図 5 では三通りの依存性解析の結果、(1) メソッドの呼び出し、(2) パラメータの計算、そして (3) メソッド `ArrayList.get()` 内部に於ける `throw` 命令の決定それぞれに対応付けられた兆候の種類と兆候に関与したオブジェクトのクラス名 [*4] を表示している。オブジェクトは (1) 複数の別名を持つ値オブジェクト或いは値保持者、(2) 隠れ更新対象、或いは (3) 期限切れの状態を提供するオブジェクトのいずれかの形で兆候に関与する。

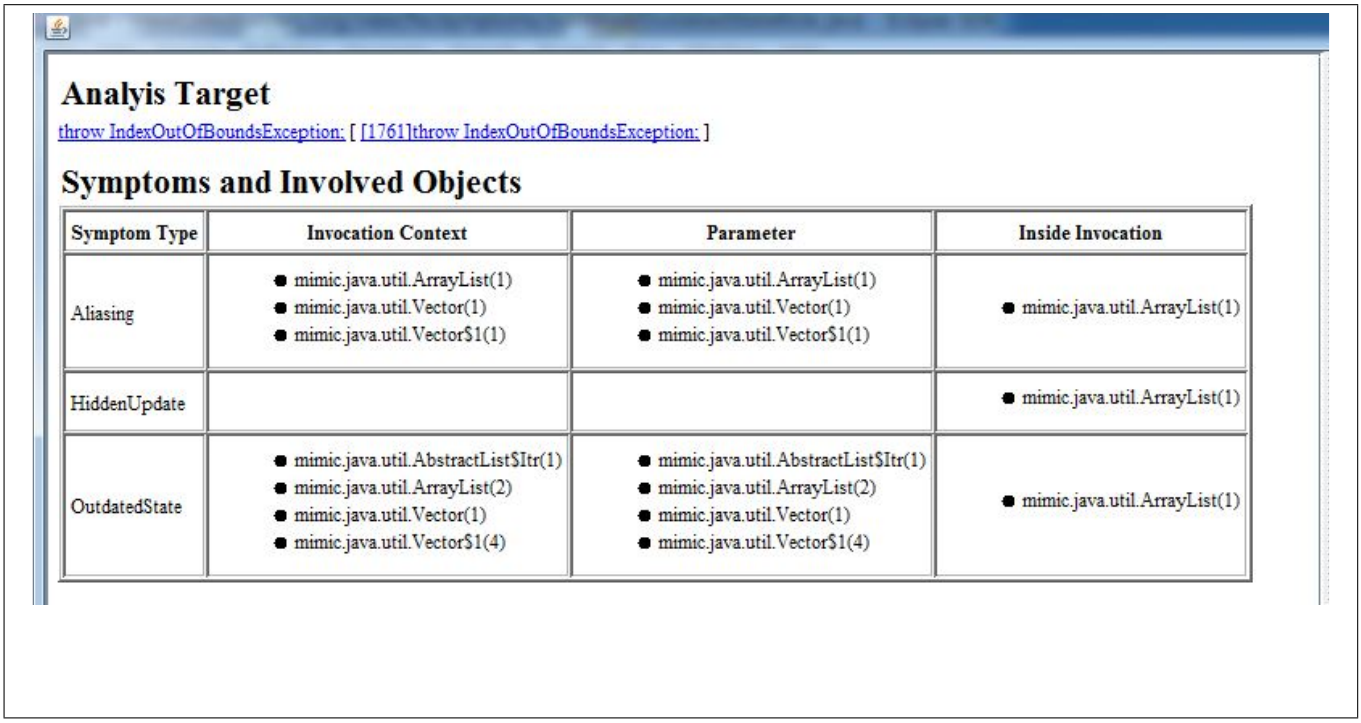
図 7 は三通りに分類された兆候を視覚的に配置したものである。メソッド呼び出しに関与した兆候は最上部に置かれている。パラメータのデータフローに関するものはそれぞれ対応するパラメータの右側に並べられている。メソッドの受け手である `this` の右並べられている兆候のうち、兆候 `Aliasing#1784` が中括弧内でくくられているが、これはこのオブジェクトが Aliasing 兆候に関与している、言い換えれば複数の別名を持っている事を意味する。その他の兆候はこれらのパラメータのデータフローの形成に間接的に関与している。メソッド呼び出し内部の制御に関する兆候は最下部に配置されている。

本試作ツールは各兆候に対して関与するオブジェクトに対してその参照経路を出力する。また実行時に兆候を実現させたソースコード行を出力する。出力結果は膨大なテキスト行から構成される。これらの出力結果の一例を図 6 に示す。図 1 のメソッド呼び出し鎖の頂点から呼び出される全メソッドのうち、174 行が兆候に関与しているが、これは手作業による調査が可能な量である。図 6 ではその一部のみが表示されている。こうした解析を可能にするために、本試作ツールは通常の手法のもの (例えば. [18]) と比べても同等あるいはそれ以上に豊富な種類のデータ構造を有するトレースを生成している。

5. 適用事例

我々は第 3 節の不具合事例に対して試作ツールによる解析を試みた。本解析実験では、例えば「`FigNode` と `FigEdge` はそれぞれ一般的なグラフの節点と辺を表現し、互いに参照関係を形成している。それぞれの廃棄処理は `dispose()` によって実装されている」というようなフレームワーククラスに関してアプリケーション開発者に公開されている知識を保守作業者が有している事を想定している。試作ツールが生成したオブジェクトの参照経路やソースコード行

[*4] ライセンス上の問題を回避するために、本試作ツールはプログラムのロード時に幾つかのライブラリクラスのバイトコードをオープンソースな Java の処理系から取得したものを置き換えている。また置き換えられたクラスのパケージ名は `mimic.` から始まるものに変更されている



Analyis Target

throw IndexOutOfBoundsException: [[1761]throw IndexOutOfBoundsException:]

Symptoms and Involved Objects

Symptom Type	Invocation Context	Parameter	Inside Invocation
Aliasing	● mimic.java.util.ArrayList(1) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(1)	● mimic.java.util.ArrayList(1) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(1)	● mimic.java.util.ArrayList(1)
HiddenUpdate			● mimic.java.util.ArrayList(1)
OutdatedState	● mimic.java.util.AbstractList$Itr(1) ● mimic.java.util.ArrayList(2) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(4)	● mimic.java.util.AbstractList$Itr(1) ● mimic.java.util.ArrayList(2) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(4)	● mimic.java.util.ArrayList(1)

図 5 兆候と関与するオブジェクト

図 7 兆候の配置

を調査する際にはこうした知識のみを用いる事にする。実験環境を構成するプラットフォームの諸元は以下の通りである：Windows PC with CPU Intel(R) Core(TM) i7-3960X CPU (3.30 GHz)、内蔵メモリ 64.0GB。Windows 7 Professional (64bit)、Eclipse SDK Version 3.7.2。

我々はまず、UML 編集プログラムを操作して図 1 のエラーを再現した。エラーの再現のために、三つの UML クラスの作成、3-項関連の形成と削除による試みに必要な数回のマウスクリックと 1 回のキーボードタイプを行なった。試作ツールが生成した実行トレースを保存し、オフライン解析を実施した。試作ツールはトレースから捕獲されていない例外投射命令を特定し、図 5 と図 7 に示された表示を実現する HTML コードを生成した。保存されたトレースの読み込みから表示までに要した時間は 4 分 57 秒である。

図 7 中のメソッド `ArrayList.get()` に付された記号 **FLib** がこのメソッドが Java ライブラリに属しており、フレームワークモードで実行されている事を示している。我々はメソッド内部の制御に三種類の兆

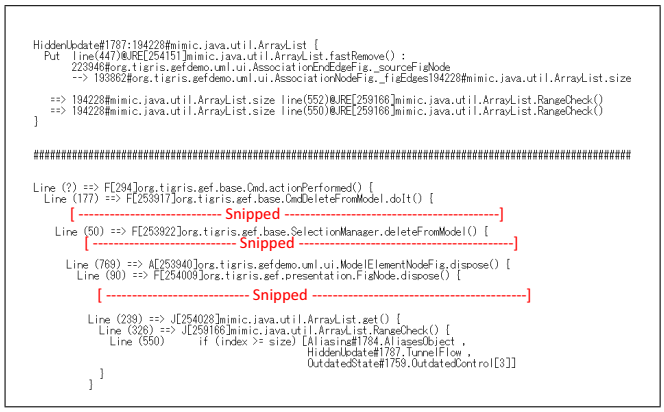


図 6 参照経路とソースコードの出力例

候 (`OutdatedState#1759`、`HiddenUpdate#1787`、そして `Aliasing#1784`) が関連付けられている点に注目した。それぞれの兆候に関与するオブジェクト *5 を調査した結果、このメソッドの受け手がメソッド内部のこれら三つの兆候全てに関与している事が判明した。

この Aliasing 兆候の出力する参照経路情報から、このオブジェクトは 3-項 UML 関連の節点からその辺を参照する `ArrayList` インスタンスである事が判明した。このオブジェクトへのある参照経路では、一旦 3-項関連の辺から 3-項関連そのものが取得され、それを経由してこのリストオブジェクトが取得されていた。アプリケーション固有側では、この参照経路に従って隠れ更新対象が取得されていた。

またこのオブジェクトへの別の二つの経路を利用して、その期限切れ状態と最新の状態が `ArrayList.get()` 内で取得されていた。結果として、他の二つの兆候 ((`OutdatedState#1759` と `HiddenUpdate#1787`) がこの Aliasing 兆候 (`Aliasing#1784`) に含まれる経路がによって実現されている事が判明した。

この Hidden Update 兆候を実施したメソッドの呼び出し関係を調査した結果、このリストオブジェクト (隠れ更新対象) に含まれている辺の一つをフレームワーク側で削除しようとする際に、アプリケーション固有部分に隠蔽される形で残りの辺も全て削除されている事が判明した。フレームワーク側では一つの辺だけを削除するつもりであるが、アプリケーション側で密かに残りの辺も削除してしまっているのである。

兆候 Outdated State の状態は `ArrayList` のインスタンス変数 `size` によって実現されている。この状態はこの兄弟辺の削除によって期限切れとなっている。Outdated State 兆候のデータフローを調査した結果、メソッド `ArrayList.get()` の引数がこの期限切れ状態を用いて計算されている事分かった。メソッド内部の条件分岐命令の中でこの期限切れの値とと更新された値が比較されており、それが例外発生の直接の引き金を引いている事が判明した。

以上の調査によって複数の参照経路を経由して参照されたリストオブジェクトに対して、アプリケーション固有部分の側でフレームワークにとって予期せぬ状態更新が実施され、それが例外発生の原因となった、すなわち副作用の発生が示唆される結果となった。図 7 の他の兆候は GUI 上の操作から対応する UML 要素を選択する処理に伴うものであり、この不具合とは直接関係しない、ノイズに相当するものと判断された。

*5 それぞれのオブジェクトには一意的な ID 番号が割り当てられている。

6. 関連研究

Zhang 等 [8] はプログラムで実行されたそれぞれの命令文に対して、誤った値を計算する誤った過程に関する寄与度をを格付けするデバッグ支援手法を提案した。彼等はこの手法によって膨大なデータ量の動的スライスを劇的に削減出来る事を証明した。彼等の研究ををさらに発展させた様々な手法が提案されている。(例えば [9], [10]) Zhang 等による手法とその後継手法はいずれも作業者による値の判定に依存している点が共通している。

上記の手法とは異なるデバッグ支援の試みとして、デバッガに問い合わせ機能を組み合わせた手法が挙げられる [11], [12] 。Lecevicius はプログラムの構成要素と意味的な制約に関する問い合わせモデルと、それに基づく問い合わせ言語を用いたデバッグ支援手法を提案している [11], [13] 。その他の有力な手法としては、Backward-In-Time Debugger [12] が挙げられる。Backward-In-Time Debugger はスタックから除去されたメソッドによる状態変更を、データフローを逆に辿る事によって特定する強力な機能を有している。

上に挙げた既存の手法は作業者がデバッグされているプログラムの詳細な実装に関して知識を有している事が前提となっている。例えばプログラムのデバッグ時に出現する局所変数値の正誤を判定するためには、その局所変数がアルゴリズムの遂行上果たす役割を知悉している必要がある。問い合わせに関しても、しばしば外部に対して隠蔽されているクラス単体の構造やクラス同士の複雑な参照関係、メソッドの呼び出しに用いられる実行時クラス等、詳細な知識無しでは効率的な問い合わせは困難であろう。

第 3 節で我々が説明したように、フレームワークアプリケーションの実行時には、フレームワークとアプリケーション固有部分のコード同士の相互呼び出しが入れ子状に発生する事が珍しくない。通常、アプリケーションのデバッグ担当者はフレームワークの実装の詳細に関する知識を有していない。こうしたフレームワークアプリケーション特有な事情によって既存の手法の適用がしばしば困難になる事が想定される。これら既存の手法を我々の提案手法と組み合わせる事によって、こうした問題の解決が期待出

来る。

7. 終わりに

本論文ではフレームワークの実装の詳細な知識を持たない作業者がそのアプリケーションをデバッグする際に、副作用の発生の有無の判断と発生過程の理解を支援する手法を提案した。本手法は副作用の発生に伴う実行時の挙動の特徴を三種類の**兆候**として形式化し、オブジェクト同士の参照関係や、実行されたソースコード行と関連付ける事によって支援を実現する点に新規性を有する。我々は提案手法を実装する試作ツールを開発し、第三者が開発した実用的なフレームワークアプリケーションで発見された不具合のデバッグ作業に適用した。我々は特定された中から複数の兆候に共通のオブジェクトが関与している事に注目した。それらの兆候を解析する事によって副作用の発生を確定した。

本試作ツールはある例外が発生した実行トレースから関連する兆候を特定し、その例外を発生させたメソッドの呼び出し、メソッドの受け手や引数、そしてメソッド内部の制御それぞれと対応する形で可視化する。これによって兆候の概要が明らかになる。試作ツールはこれに加えて兆候に関与するオブジェクトや、兆候を実現したソースコードをテキスト形式で出力している。デバッグ作業でより効率的に兆候を利用するためには、これらの情報の洗練された可視化が必要となる。

本提案手法の一般性を検証するために、現在我々は副作用が疑われる、或いは無関係だと思われる不具合事例を収集している。不具合事例に対して本手法を適用する、第三者が参加した検証実験を今後実施する予定である。我々の試作ツールでは既存の多くの動的解析手法と比べて同等或いはより豊富なデータ構造を有するトレースを生成している。これによってツールによる処理速度が犠牲になっている。実用性の点からこうした効率に関する向上も重要な将来課題である。

8. Acknowledgments

研究の方向性に関して重要な示唆を下さった萩田紀博教授に感謝致します。本研究は MEXT/JSPS 科研費 [挑戦的萌芽 (No.23650016)、基盤 (C)(No.24500079)、及び基盤 (B) (No.23300009)] の助成を受けています。

参考文献

[1] Fayad, M. E., Schmidt, D. C. and Johnson, R. E.: *Building Application Frameworks*, John Wiley & Sons. (1999).

[2] Shull, F., Lanubile, F. and Basili, V. R.: Investigating Reading Techniques for Object-Oriented Framework Learning, *IEEE Transactions on Software Engineering*, Vol. 26, No. 11, pp. 1101–1118 (2000).

[3] Heydarnoori, A., Czarnecki, K., Binder, W. and Bartolomei, T. T.: Two Studies of Framework-Usage Templates Extracted from Dynamic Traces, *IEEE Transactions on Software Engineering*, Vol. 38, No. 6, pp. 1464–1487 (2012).

[4] Monperrus, M., Bruch, M. and Mezini, M.: Detecting Missing Method Calls in Object-Oriented Software, *ECOOP*, pp. 2–25 (2010).

[5] Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern, http://www.martinfowler.com/articles/injection.html.

[6] Kume, I., Nakamura, M. and Shibayama, E.: Toward Comprehension of Side Effects in Framework Applications as Feature Interactions, *he 19th Asia-Pacific Software Engineering Conference (APSEC 2012)* (2012).

[7] Agans, D. J.: *Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems*, AMACOM (2002).

[8] Zhang, X., Gupta and Gupta, R.: Pruning Dynamic Slices with Confidence, *Conference on Programming language design and implementation*, ACM, pp. 169–180 (2006).

[9] Zhang, X., Gupta, N. and Gupta, R.: Locating faults through automated predicate switching, *Proceedings of the 28th international conference on Software engineering*, ICSE '06, New York, NY, USA, ACM, pp. 272–281 (online), DOI: 10.1145/1134285.1134324 (2006).

[10] Alves, E., Gligoric, M., Jagannath, V. and d'Amorim, M.: Fault-localization using dynamic slicing and change impact analysis, *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, Washington, DC, USA, IEEE Computer Society, pp. 520–523 (online), DOI: 10.1109/ASE.2011.6100114 (2011).

[11] Lencevicius, R., Hölzle, U. and Singh, A. K.: Dynamic Query-Based Debugging of Object-Oriented Programs, *Automated Software Engineering*, Vol. 10, No. 1, pp. 39–74 (2003).

[12] Hofer, C., Denker, M. and Stéphane Ducasse: Design and Implementation of a Backward-In-Time Debugger, *Proceeding of NODe 2006*, Lecture Notes in Informatics, Vol. P-88, pp. 17–32 (2006).

[13] Lencevicius, R.: *Advanced Debugging Methods*, Kluwer Academic Publishers (2000).

[14] Hogg, J.: Islands: Aliasing Protection In Object-Oriented Languages, *OOPSLA*, pp. 271–285 (1991).

[15] Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley (1994).

[16] Lienhard, A.: *Dynamic Object Flow Analysis*, Lulu.com (2008).

[17] Meyer, B.: *Object-Oriented Software Construction*, Prentice-Hall, second edition (1997).

[18] Wang, T. and Roychoudhury, A.: Using Compressed Bytecode Traces for Slicing Java Programs, *International Conference on Software Engineering*, IEEE, pp. 512–521 (2004).