Vol.2012-SE-175 No.13
2012/3/15

トレース解析手法を利用した逸脱コードの特定

久米 出[†1] 柴山悦也[†2]

近年多くのオブジェクト指向アプリケーションがアプリケーションフレームワークの上に構築されている中で、フレームワークの利用に関する約束事を守らない**逸脱コード** *(deviant code)* の問題が注目されるようになってきた。本論文ではアプリケーション側からフレームワーク側のメソッドを誤った方法で呼び出し、フレームワーク内部で**逸脱挙動** *(deviant behavior)* を発生させる逸脱コード特定を支援するトレース解析手法を提案する。我々の手法では逸脱コードの実行よって引き起こされるフレームワークの内部挙動を抽象化して呈示する点に特徴を持つ。実際に第三者が開発したフレームワークアプリケーション中の逸脱コードを特定する事例を通じて本手法の有効性を議論する。

Deviant Code Identification by Trace Analysis

Izuru Kume[†1] and Etsuya Shibayama[†2]

Recently more and more object-oriented applications are build on application frameworks, and the problem of *deviant code* to use a framework in an incorrect way becomes more serious. In this paper we propose a trace analysis approach to support identification of a kind of deviant code that invokes framework methods incorrectly and causes a *deviant behavior* inside of a framework. The novelty of our approach is in a kind of abstract view that shows how a deviant code execution causes a deviant behavior. We discuss the usefulness of our approach by through a case study to cope with a deviant code example in a framework application developed by a third party.

†1 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
†2 東京大学
The University of Tokyo

1. はじめに

近年、オブジェクト指向アプリケーション開発に於けるアプリケーションフレームワークの利用が進むとともに、アプリケーション側がフレームワークの正しい用法を守らない、所謂**逸脱コード** *(deviant code)*[1] の問題が注目され始めている。逸脱コードという概念は元々はオブジェクト指向アプリケーションフレームワークに限定された概念ではない。逸脱コードはシステム固有或いは一般的なコーディング規則違反を実装し、**逸脱挙動** *(deviant behavior)*[2] によってその存在を明らかにする。一般に逸脱挙動から因果関係を辿って逸脱コードを発見する事は一般に困難である。よって従来の手法では動的あるいは静的なパターン化や統計的な手法を適用して逸脱コードの発見を支援している[1),3)–7)]

フレームワークアプリケーションを対象とした実証的な研究としては Monperrus 等[1] による *Missing Method Calls* の発見手法が挙げられる。これはアプリケーション固有のプログラムコードからフレームワーク側のコードの呼び出しに関する不具合を対象とした手法である。フレームワークの基本概念を説明する学術的な文献[8)–10)] ではフレームワークの拡張[8),9)] やフレームワーク側からのアプリケーションコードの呼び出し (制御の反転)[10)] に関して説明される事が多い。しかしながら現実にフレームワークを利用する際にはフレームワークの初期化や Java 言語の `super` メソッド呼び出し見られるようにアプリケーション側からフレームワーク側のメソッド呼び出しが必要となる場合が多い。Monperrus 等[1] による調査によって、こうした呼び出しに関連する逸脱コードがアプリケーション開発の現場でしばしば発生する事が明らかにされている。

上記の手法を適用するためには、逸脱コードか否かを特定するための事例をある程度集める必要がある。例えば Monperrus 等[1] の手法では Eclipse 開発ページで公開されたバグデータベースから、アプリケーションからフレームワークの特定のメソッド呼び出し事例を収集している。しかしながら、アプリケーション開発の現場ではこうした前提が常に満たされるとは限らない。フレームワークが Eclipse のように広く用いられていない場合はそもそも数多くの呼び出し事例を収集する事は困難である。さらに、こうした従来の手法では発見された逸脱コードがどのように逸脱挙動を実現するのかを明らかにしない。Monperrus 等[1] は呼び出し事例が多数収集されている場合でも、逸脱コードを訂正するためにプログラムコードの調査が必要になる事例の存在を報告している。

我々は逸脱コーロの発見ではなく、発現した逸脱挙動からの**逸脱コードの特定**を研究の目標としている。研究の現段階では Java で記述されたフレームワークアプリケーションが

Vol.2012-SE-175 No.13
2012/3/15

副作用を発生させる種類の逸脱コードに焦点を当てている。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プログラムを対象とする。我々の手法では逸脱挙動の実行トレースを解析し、その挙動を抽象化する事によって逸脱の原因となった副作用とその発生の原因の特定を支援する。実行トレースは我々が過去に開発した Java byte code の instrumentation ツール[11),12)] を用いて取得する。

本節の最後に本論文の以降の構成を説明する。第 2 節では第三者が開発した実用的なフレームワーク上に構築されたアプリケーションとその不具合事例を説明する。第 3 節で我々の手法を説明する。上記事例への適用を第 4 節で説明し、適用した結果を第 5 節で議論する。第 6 節で関連研究について述べ、第 7 節で結びの言葉を述べる。

2. 逸脱挙動事例

本論文では第三者が開発した実用的なアプリケーションフレームワーク GEF(Graph Editing Framework)[13)⋆1] 上の例題アプリケーション GEFDemo[14)] の不具合事例を取り上げる。GEF はノードとエッジから構成されたグラフ図形を編集する処理の骨組を提供するオープンソースのフレームワークであり、実用的な UML 編集ソフトウェア[15)] の開発に用いられている。GEFDemo は GEF の利用者の学習のために、GEF 開発者によって作成された例題アプリケーションであり、簡単な UML 図式の編集機能が実装されている。

以降ではあるクラスがフレームワーク (GEF) 側で定義されているのか、アプリケーション (GEFDemo) 側で定義されているのかを特定出来るものとする。実際、クラスのパッケージ名から容易に判別が可能である。フレームワークの核心クラス[16)] の役割や、フレームワークのホットスポットとして設計されているメソッド、アプリケーション側から呼び出されているフレームワーク側のメソッドの処理について問題領域上の概念 (この場合はグラフの結点と辺) を用いて説明出来る事を仮定する。例えばクラス `FigNode` は矩形の輪郭を持つグラフの結点図形を表現するクラスであり、そのメソッド `dispose()` はこのような種類の結点を消去する際に必要な処理を実装している。

GEFDemo を用いて UML 図式を編集した実行例を図 1 に示す。まず図の左上に示すように三つの UML クラス図形を一つの多対多関連で結合する。その後で関連を選択して削除する。関連を削除した時点で図の下部に示すような例外が発生し、図の右上に示すように関連が削除されずに残ってしまう。

⋆1 Eclipse の同名のフレームワークとは別物である。

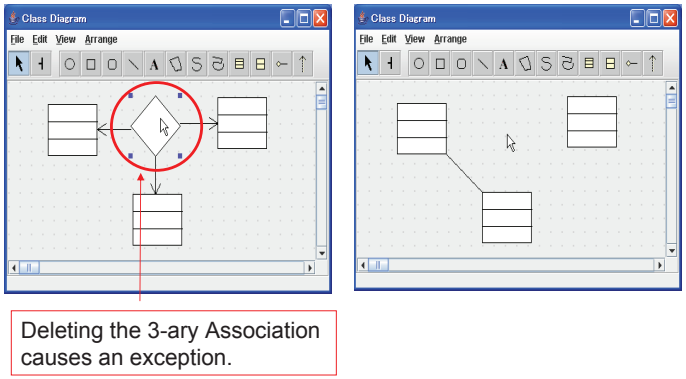

図 1 UML 図式の編集時のエラー

上の図にエラー表示に示される例外発生に至るメソッド呼び出し文脈と開発環境 (Eclipse) のツールを用いた調査の結果、以下のような事柄が明らかにされる：

- 図 1 のクラスを表現する図形は GEFDemo クラスによって実装されており、このクラスは GEF の図形クラスを継承している
- 上記 GEFDemo の図形クラスには自身の削除処理を実装したと思われるメソッドが定義されている
- 上記メソッドから `super` 呼び出しを用いて GEF で定義されたコードが呼び出されており、実質的な削除処理は GEF のコードによって実行されている
- 例外は GEFDemo から呼び出された図形削除処理メソッドの実行中に発生している

フレームワークアプリケーションの設計と挙動を説明する文献[9)] に説明されるように、上記の実行でもフレームワーク (GEF) 側の Template メソッドからアプリケーション (GEFDemo) 側の Hook メソッドが呼び出されている。しかしながら呼び出された Hook メ

ソッドから super を用いてフレームワーク側のメソッドが呼び出されている。一般に GEF は全ての図形作成の際にアプリケーションから隠蔽された一連の初期化を実行しており、図形の削除時にもこれに対応した処理が呼び出される必要がある。

以上の事柄からアプリケーション側 (GEFDemo) から誤った方法でフレームワーク (GEF) 側のメソッドが呼び出された事によってフレームワーク内部でエラーが発生した、則ち我々が対象としている逸脱挙動が発生している可能性が高いと判断される。

GEFDemo のコード中には上のように super 呼び出しを用いて UML モデル図形の削除処理を成功させている例が存在する。しかしながら単に自分自身の中で上位クラスのメソッドを呼び出しているだけであるために、メソッド呼び出しやオブジェクトのやりとりのパターンを抽出し、比較する第 1 節で紹介した既存手法の適用は困難である。

実際のところこの逸脱挙動の発生の仕組みを理解するためには、フレームワーク内部で実行される複雑な処理過程を辿る必要がある。この処理過程は入り組んだオブジェクト同士の参照、参照されたオブジェクトに対するメソッド呼び出し、メソッド呼び出しによって引き起こされる状態変更、状態が変更されたオブジェクトへの参照とメソッドの呼び出しが含まれている。

逸脱コードを特定し、不具合を修正するためにはフレームワーク内部の挙動を何らかの形で理解する必要がある。しかしながら、上述した複雑な内部挙動を例えばデバッガを用いて追うような作業は多大な労力を必要とする上に、フレームワークを利用する主旨からも避けるべきである。少ない労力で必要な情報を取得するための手法が必要とされている。

3. 提 案 手 法

3.1 概　　要

我々は逸脱挙動のトレースからその発生の仕組みを抽象化してアプリケーション開発者に提示する手法を提案する。提案手法の概略を図 2 に示す。まず、これまで我々が開発した Java の byte code instrumentation ツール[11),12)] を適用し、プログラムの実行トレースを取得する。我々のツールを用いて取得されたトレースは豊富なデータ構造を有している。トレースには以下の情報が含まれている：

- オブジェクトと基本値双方のデータの流れ
  - オブジェクト同士の参照関係の変遷
  - 基本値の生成、演算、オブジェクトからの参照
- メソッド間の呼び出し関係
- 条件分岐による制御の流れ
- 制御の流れの形成に於けるデータ依存性
  - メソッドの受け手として参照されるオブジェクト
  - 分岐条件から参照される値

データの流れはオブジェクトのインスタンス変数や配列に対する値の代入と参照、局所変数に対する値の変更、instanceof 演算子や数値演算による基本値の計算、クラスインスタンスや配列の作成、定数値やメソッドによる返り値によって形成される。こうしたデータの流れの生成に関与する行為、或いはメソッド呼び出しや条件分岐はトレース中の**操作** *(Action)* として表現される。各操作はそれが実行されたメソッド呼び出しと対応付けられている。データの流れから各操作 にはデータ依存性が定義される。

上に述べたようものと同様なデータ依存性は Java プログラムを対象としたスライシングのために生成されるトレース中にも表現されている。[17)] しかしながら、我々のツールが生成するトレースには単なる依存関係だけでなく、上記の個別のメソッド呼び出し、操作、データそのものが Java のオブジェクトとして表現されている。実際のところ、我々のツールによって生成されるトレースは操作とデータから形成される制御の流れとデータの流れが複雑に絡み合う有向グラフとして実現されている。

我々は生成されたトレースから開発者の目的に必要な箇所のみを特定し、その内容を抽象化して表示する事を目的としている。抽象化によって作成された表示内容を本論文では *Object-Effect View* と呼ぶ。一般に dynamic slicing 等の用途で生成されるトレースのデータ量は膨大であり[18)]、我々のツールが生成するトレースもその例外では無い。我々はトレースに対する検索と *Object-Effect View* 定義によってデータ量の問題に対処しようと試みている。また解析作業の効率化のために検索や Object-Effect View の作成の作業工程のパターン化を追求する。

3.2 抽象化の方針と技法

3.2.1 対象の限定

本節では第 3.1 節で述べた Object-Effect View 作業のパターン化の実現に向けた議論を行う。議論の焦点を絞るために本論文ではトレース解析の対象となるプログラムと挙動、及び解析の目的を以下のように限定する。トレース解析の対象をフレームワーク上に構築されたアプリケーションとする。また、アプリケーション側からフレームワーク側のメソッドを呼び出した副作用に起因するフレーワーク内部の逸脱挙動を解析対象とする。解析の目的は現象として現れた逸脱挙動を引き起こした副作用の特定とする。

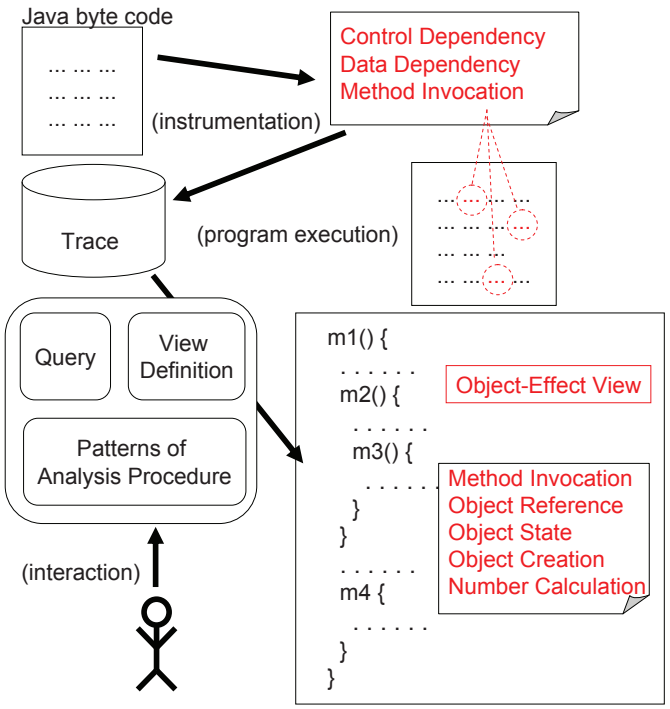

図 2 トレース解析の流れ

3.2.2 オブジェクト参照と状態変更の経緯

まず抽象化による結果、則ち Object-Effect View の表示形式に関して検討する。逸脱挙動からその原因を特定する作業はデバッグの一種であると考えられる。デバッガを用いた通常のデバッグ作業ではエラーが発生した時点とそのメソッド呼び出しの文脈が最初に表示される。しかしながら、エラーの原因となるプログラムコードは、メソッド呼び出しによって積まれたスタックには含まれていない事が多い。「この変数の値を最後に設定した箇所」を特定するために作業者は自身の仮説に基づいてブレークポイントを設定し、プログラムの再実行を繰り返しながらエラーの原因箇所と特定しようとする。

こうした「最後に変数の値を設定した箇所」の特定を支援する手法として、Omniscient debugger[19] や Backward-In-Time Debugger[20] が開発されている。しかしながら、これらの技術は我々が第 2 節で紹介した事例の本質的な問題解決にはあまり役に立たない。実際、我々の事例で考えるとオブジェクトの状態を変更しているフレームワーク側のメソッドの実装に誤りがあるわけではない。本来状態が変更されるべきでないオブジェクトがそのメソッドに渡されて状態が変更されてしまった事が問題なのである。本事例の問題を解決するためには、該当オブジェクトとその状態がフレームワーク側で変更され、変更された状態の参照に至る一連の経緯が明らかにされるべきである。

我々はこうした経緯、則ちオブジェクトに対する一連の操作こそが逸脱挙動の実体であり、この過程を開始したアプリケーション側のコードこそが逸脱コードとして特定されるべきであると考えている。

3.2.3 メソッドの呼び出し構造

第 3.2.2 節で述べたように、あるオブジェクトに対する状態の変更と、その状態の参照は異なるメソッド呼び出し木の下に実行される事が珍しくない。また、状態が変更されたオブジェクトが別のオブジェクトのインスタンス変数値として参照されたり、或いはメソッド呼び出しの返り値として取得される可能性も考えられる。一般に逸脱挙動を構成する一連の操作を実行したメソッドは枝分れの多い呼び出しを形成する傾向にある。解析の最初の段階から複雑なメソッド呼び出し木を表示するのは避けるべきであると我々は考えている。

解析の始めに提示すべきメソッド呼び構造に関する我々の基本的な主張を図 3 に示す。我々は逸脱挙動が (例えば例外の発生によって) 明らかになったメソッドへの呼び出しから構成される、線型な木構造に近い形から出発すべきであると考える。図 3 では例外の発生に至るメソッド呼び出し文脈と、その呼び出しに伴うオブジェクトの受渡しが示されている。このオブジェクトは呼び出しの上から三つめのメソッドから呼び出された別の呼び出しの枝

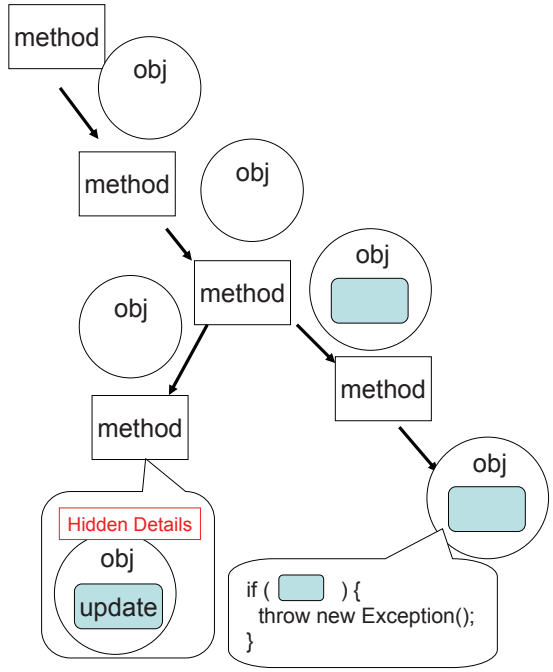

図 3 線型な木構造

の中で状態が変更されている。このような呼び出しの別の枝は始めからその詳細を表示するのではなく、枝の内部の実行内容を抽象化して表示すべきである。結果として、本来のそれよりも短い枝が線型の木構造に追加される事になる。このように最初に表示される線型のメソッド呼び出し構造を**幹**、本来よりも短かく表示される過去のメソッド呼び出しの枝を**瘤**と呼ぶ事にする。

3.2.4 メソッド実行内容の抽象化

メソッド呼び出しの瘤として表現されているメソッドの実行内容は幹に属するメソッドよりも高い抽象度で表現されるべきである。我々は瘤の内部のみで参照されるオブジェクトが関与する操作を表示しない方針を採用する事によってこうした要求に答えられると考える。この方針に従えば、瘤として表現されるメソッドの受け手、引数、返り値、さらに瘤の外で明らかにされているインスタンス変数への代入値として参照されるもの以外のオブジェクトが関与する操作は全て隠蔽される。

このように抽象化の方針を定める事によって、解析作業中に瘤が展開され、そのメソッド呼び出し関係が明示化される度合に対応してメソッドの実行内容が詳細化されると期待出来る。

こうした抽象化の度合の調整は幹に属するメソッドに関しても適用されるべきである。通常 Java のライブラリクラス、例えば Collection Framework に属するクラスのメソッド実行の詳細は一切明らかにされるべきでないだろう[⋆1]。逸脱コードを特定するためにはアプリケーション固有クラスのメソッド実行の詳細な内容が必要になるかもしれないが、フレームワークの内部挙動に関しては出来るだけ詳細化されないまま解析の目的が達成される事が望ましい。我々は、フレームワーク内部でのメソッド呼び出しに関して部分的に瘤化する事が望ましいと考えている。

4. 手法の適用事例

本節では第 3 節で説明した抽象化を第 2 節の例題に適用した結果を示す。現在の実装では抽象化のために作成されたデータを表示する処理が一部未完成であるため、以降で示される表示結果の一部は手動で作成されている。

⋆1 我々の instrumentation ツールは Open JDK のソースコードからバイトコードを取得し、クラスのパッケージ名を変更する事によって、Sun のライセンスに違反する事無く Java の List、Set、Map の内部挙動のトレースを取得している。厳密には Sun のクラスではなく、Open JDK のクラスのメソッドの実行トレースが取得されている事になる。

解析作業の第一歩として例外が発生したメソッド呼び出し文脈を特定する。そのために図 1 で表示されているエラーメッセージより発生した例外クラス (IndexOutOfBoundsException) を特定し、この例外クラスのインスタンスを throw する操作を検索する。検索が成功すると、各操作毎にメソッド呼び出しの幹表示される。我々の例題ではこのような例外発生は一度だけ実行されるため[*1]以下のような幹が一つだけ表示される。

```
F[1]: Cmd.actionPerformed()
{
 F[2]: CmdDeleteFromModel.doIt()
 {
  F[3]: SelectionManager.deleteFromModel()
  {
   A[4]: ModelElementNodeFig.dispose()
   {
    F[5]: FigNode.dispose()
    {
     J[6]: ArrayList.get()
     {
      J[7]: ArrayList.RangeCheck()
      {
       [0] throw IndexOutOfBoundsException
      }
     }
    }
   }
  }
 }
}
```

我々が実装している解析ツールは検索された例外発生操作と、それに至る幹を構成するメソッド呼び出しに対して一意的な数値が ID として割り当てる。ここでは例外発生に対しては 0 が、それに至るメソッド呼び出しは呼び出し順に 1 から 7 までの数値が ID として割り当てられている。一旦 ID が割り当てられた操作はその ID 値によって指定する事が可能になる。各メソッド呼び出しの ID 値の前に表示されている記号はメソッドの種類は示している。アプリケーション固有のクラスで定義されたメソッドには A、フレームワークのメソッドに対しては F が付与される。J は Java ライブラリクラスのメソッドを意味する。ここでは ArrayList に対してメソッド get() の内部で呼び出される

例外の発生文脈が特定出来たので、幹の中に瘤を作成する。作業者が特に指定しなければ、フレームワークメソッド同士の呼び出しとライブラリのメソッド同士の呼び出しが隠蔽される。ここで、7 番目のメソッド (J[7]) の内部で例外の発生を引き起こした分岐 (もし有れば) の条件として参照された値が ArrayList.get() の外側の視点で表示する。その結果を以下に示す。

```
F[5]: FigNode.dispose()
{
 A[9]: AssociationEndEdgeFig.dispose()
 {
  F[10]: FigNode.delete()
  {
   [8] ArrayList.size = 0;
  }
 }
 [15] 1 <-- iinc2 ... Line:238
 [16] ArrayList <-- AssociationNodeFig._figEdges ... Line:239
 J[6]: ArrayList.get()
 {
  [0] throw IndexOutOfBoundsException
 }
}
```

表示結果を簡略にするために、ここでは番号 4 番以上のメソッドを省略する。実際のツールの出力では 1 番から 4 番までのメソッド呼び出しが表示されている。上の出力結果には以

⋆1 このアプリケーション (GEFDEmo) の初期化処理の中で複数が例外が発生し、アプリケーション内で catch されている。

前には表示されていなかった操作が三つ追加されており、それぞれ番号 8、15、16 が割り振られている。これらは順に ArrayList のインスタンス変数 size に対する値の代入、++ 演算子による計算結果、クラス AssociationNodeFig のインスタンス変数_figEdges の値の取得を意味している。ここでフレームワーク側メソッド F[10] は瘤として表現されている。瘤を展開した結果を以下に示す。

```
F[5]: FigNode.dispose()
{
 A[9]: AssociationEndEdgeFig.dispose()
 {
  F[10]: FigNode.delete()
  {
   J[13]: ArrayList.remove()
   {
    [8] ArrayList.size = 0;
   }
  }
 }
 [15] 1 <-- iinc2 ... Line:238
 [16] ArrayList <-- AssociationNodeFig._figEdges ... Line:239
 J[6]: ArrayList.get()
 {
  [0] throw IndexOutOfBoundsException
 }
}
```

上の出力結果には ID 値の欠番が生じている。実は瘤を展開したメソッド呼び出しの枝にはフレームワークメソッド同士の呼び出しと、ライブラリメソッド内部の呼び出しが瘤として隠されている。欠落した ID 値はこれらのメソッド呼び出しに割り振られている。

メソッド呼び出し木をここまで展開した段階で、各メソッドの受け手や引数として現れるオブジェクトに ID 値を与え、それらの相互参照や状態変更の過程を表示する。

```
F[5]: FigNode#18.dispose()
{
Reference
{
 #19 <-- this(#18)
}
A[9]: AssociationEndEdgeFig#19.dispose()
{
 Reference
 {
  #18 <-- this(#19)
 }
 F[10]: FigNode#18.delete()
 {
  Reference
  {
   #17 <-- #19 <-- this(#18)
  }
  J[13]: ArrayList#17.remove()
  {
   Reference
   {
    this(#17){?}
   }
   Update
   {
    this(#17){[8] <update> this(#17{?}}
   }
  }
 }
}
Reference{
  [16] #17{![8]} <-- this(#19)._figEdges
```

```
 }
 J[6]: ArrayList@[16].get()
 {
  Reference{
   this(ArrayList#17){![8]}
  }
  [0] throw IndexOutOfBoundsException <== ![8]
 }
}
```

上の表示の中で#の後に続く数値はオブジェクトに与えられた ID 値である事を意味する。実際この表示中ではメソッドの受け手となるオブジェクトが特定されている事に注意して欲しい。上の表示中に表れた `Reference` 節はメソッドに与えられたオブジェクトからどのような経路で値が参照されるのかを示している。`Update` 節はメソッドの実行中に発生したオブジェクトの状態の変更 (インスタンス変数に対する値の代入) を明示するものである。メソッドの受け手となるメソッドは `this` として表示されている。

上の表示結果から判明した事実を以下に説明する。まず、FigNode のインスタンス`#18` を消去するために `dispose()` が呼び出されている。このメソッドはクラス図式 1 上に表示されている多対多関連図形の削除に対応してい呼び出されていると考えられる。ここでこの `FigNode` オブジェクトのインスタンス変数からオブジェクト`#19` が参照され、インスタンス変数値として取得されたオブジェクト (`AssociationEndEdgeFig`) に対してメソッド `dispose()` が呼び出される。これはフレームワークの Template メソッドからアプリケーションの実装する Hook メソッドに対する呼び出しである。

アプリケーション側メソッド `A[9]` の中で再びオブジェクト`#18` に対するメソッドが呼び出されている、このメソッド `F[10]` の中でオブジェクト`#19` を経由して `ArrayList` インスタンス`#10` が参照され、メソッド `remove()` が実行され、その結果このインスタンスの内部状態が変更されている。この出力では `ArrayList` の内部の構造は抽象化されている事に注意して欲しい。

メソッド `A[9]` の return 後、メソッド `F[5]` で `ArrayList` インスタンス`#17` が参照され、メソッド `get()` が呼び出されている。この時参照されているオブジェクトの内部状態が状態変更を表す操作 `[8]` によって変更されている。この操作によって変更された状態から条件分岐 (表示されていない) の条件が計算され、例外の発生を引き起こしている。

以上の結果より、我々はアプリケーション固有のメソッド `A[9]` 中からフレームワーク側メソッド `F[10]` に対する呼び出しを逸脱コードとして特定し、その結果引き起こされる一連のオブジェクトの参照と状態変更、変更された状態が引き起こす例外の発生を逸脱挙動として特定する。

5. 議　　論

我々が解析の対象とした GEF フレームワークは行数が 43,000 行以上、クラス数は 251 にのぼる。大規模ではないが、実用的な規模のプログラムであると言える。このような実用的なプログラムのトレース解析の場合、トレースデータ量が膨大になる。膨大なトレースから問題解決に必要な部分を取り出す事が可能であったのは、我々が対象とするプログラムの種類を特定し、かつ抽象化の方針を適切に設定したからであると思われる。またトレースの豊富なデータ構造を利用した検索機能も実用上重要であると思われる。実際、我々の適用事例でも例外の発生の原因となる条件を計算するデータをソースコードの参照を行う事無く特定している。

本論文で紹介した事例では実行中にループが何度か発生したもののの、繰り返し回数が少ない事も適用が成功した要因の一つであると考えられる。ループの実行毎に副作用が蓄積するような場合に適切な抽象化の方針を考える必要があると思われる。また、現在の実装では実行効率に関する工夫が為されていないため、ツールから解析対象プロセスを起動するために数分必要とされている。このような実行効率の改善も実用性の面から極めて重要であると思われる。

6. 関 連 研 究

我々の提案手法は膨大なトレース情報から逸脱挙動を形成する箇所を特定し、その過程を抽象化して提示する点に特徴がある。特定のオブジェクトに注目したトレース解析手法としては Quante による Dynamic Object Process Graphs[21),22)] が挙げられる。

オブジェクト間の参照関係に焦点を当てた先駆的な解析手法として Object Flow Analysis[23),24)] が挙げられる。Object Flow Analysis は従来のオブジェクト指向プログラムの依存性[25)] をさらに拡張しようという動機から出発したものである。依存性解析の回帰テスト[26)] や Back-In-Time デバッガの効率化[27)] の分野では成功を収めている。

しかしながら、可視化に関してはそれほど成功しているとは思えない。オブジェクトの参照構造の変遷を参照グラフの時系列的な変化として可視化しようとしているが実用的なプ

ログラムの実行結果を表現するグラフはノード数が数百単位に達っしてしまい、これをプログラム理解に直接利用する事は困難であると考えられる[24)]。Object Flow Analysis は解析の対象や目的を特定しない、一般の解析手法を追求しているためにこのような表現方法を追求したと考えられる。

7. 終わりに

本論文ではアプリケーション固有のコードからフレームワーク側のメソッドの呼び出しを行う逸脱コード、特にフレームワークの内部状態を誤った状態に変更し、逸脱挙動を実現する種類の逸脱コードの特定問題を取り扱った。フレームワークの内部挙動を抽象化して表現する事により、エラーを引き起こす逸脱挙動の過程の理解と逸脱コードの特定を支援するトレース解析手法を提案した。提案手法を実際のフレームワークアプリケーションに適用し、その実用性を議論した。

謝辞 抽象化のあり方に関して助言下さった大阪芸術大学の武村泰宏先生、甲南大学の新田直也先生、及び株式会社アーヴァイン・システムズの皆様に謹んで感謝の意を表します。

参考文献

1) Monperrus, M., Bruch, M. and Mezini, M.: Detecting Missing Method Calls in Object-Oriented Software, *ECOOP*, pp.2–25 (2010).
2) Engler, D., Chen, D. Y., Hallem, S., Chou, A. and Chelf, B.: Bugs as deviant behavior: a general approach to inferring errors in systems code, *Symposium on Operating Systems Principles*, ACM, pp.57–72 (2001).
3) Hangal, S. and Lam, M. S.: Tracking Down Software Bugs Using Automatic Anomaly Detection, *International Conference on Software Engineering*, IEEE, pp. 291–301 (2002).
4) Wasylkowski, A., Zeller, A. and Lindig, C.: Detecting Object Usage Anomalies, *ESEC/FSE*, pp.35–44 (2007).
5) Li, Z. and Zhou, Y.: PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software, *ESEC/FSE*, pp.306–315 (2005).
6) Dallmier, V., Lindig, C. and Zeller, A.: Lightweight Defect Localization for Java, *ECOOP*, pp.528–550 (2005).
7) Livshits, B. and Zimmermann, T.: DynaMine: Finding Commong Error Patterns by Mining Software Revision Histories, *ESEC/FSE*, pp.296–395 (2005).
8) Johnson, R. E. and Foote, B.: Designing Reusable Classes, *Journal of Object-Oriented Programming*, Vol.1, No.2, pp.22–35 (1988).
9) Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley (1994).
10) Sparks, S., Benner, K. and Faris, C.: Managing Object-Oriented Framework Reuse, *IEEE Computer*, Vol.29, No.9, pp.52–61 (1996).
11) Kume, I. and Shibayama, E.: A Conceptual Model for Comprehension of Object-Oriented Interactive Systems, *International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)* (2009).
12) Kume, I. and Shibayama, E.: Feature Interactions in Object-Oriented Effect Systems from a Viewpoint of Program Comprehension, *International Conference on Feature Interactions* (2009).
13) : GEF Project Home Page, http://gef.tigris.org/.
14) : gefdemo Project, http://gefdemo.tigris.org/.
15) : ArgoUML Project Home Page, http://argouml.tigris.org/.
16) Fayad, M.E., Schmidt, D.C. and Johnson, R.E.: *Building Application Frameworks*, John Wiley & Sons. (1999).
17) Wang, T. and Roychoudhury, A.: Using Compressed Bytecode Traces for Slicing Java Programs, *International Conference on Software Engineering*, IEEE, pp. 512–521 (2004).
18) Tip, F.: A survey of program slicing techniques, *Journal of Programming Languages*, Vol.3, pp.121–189 (1995).
19) Lewis, B.: Debugging Backwards in Time, *International Workshop on Automated Debugging (AADEBUG)* (2003).
20) Hofer, C., Denker, M. and Stéphane Ducasse: Design and Implementation of a Backwrd-In-Time Debugger, *Proceeding of NODe 2006*, Lecture Notes in Informatics, Vol.P-88, pp.17–32 (2006).
21) Quante, J. and Koschke, R.: Dynamic Object Process Graphs, *European Conference on Software Maintenance and Reengineering*, IEEE, pp.81–90 (2006).
22) Quante, J.: Online Construction of Dynamic Object Process Graphs, *European Conference on Software Maintenance and Reengineering*, IEEE, pp.113–122 (2007).
23) Lienhard, A., Ducasse, S., Gîrba, T. and Nierstrasz, O.: Capturing How Objects Flow at Runtime, *International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pp.39–43 (2006).
24) Lienhard, A.: *Dynamic Object Flow Analysis*, Lulu.com (2008).
25) Salah, M. and Mancoridis, S.: A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions, *International Conference on Software Maintenance*, IEEE, pp.72–81 (2004).
26) Lienhard, A., Greevy, O. and Nierstrasz, O.: Tracking Objects to Detect Feature Dependencies, *International Conference on Program Comprehension*, IEEE, pp.59–

Vol.2012-SE-175 No.13
2012/3/15

68 (2007).
27) Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugger, *ECOOP*, pp.l592–615 (2008).