Vol.2013-SE-179 No.20
2013/3/11

フレームワークアプリケーションの抽象化のための動的解析手法

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

概要：近年のソフトウェア開発に於けるアプリケーションフレームワークの普及と共に、その正しい利用法を効率的に学ぶ手法がますます重要になっている。フレームワークの学習の障害として、その複雑性と、所謂制御の反転 (Inversion of Control) に特徴付けられる独特な実行形態が挙げられる。我々はこれらの障害を克服するために、フレームワークアプリケーションの内部挙動を抽象化して表現する機能モデル (feature model) と、動的解析を用いたモデリングを提案する。本論文では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内で発見されたフレームワーク利用の誤りを事例として我々の取り組みとその将来課題を説明する。

1. はじめに

近年多くのアプリケーション開発で様々なフレームワークが利用されるようになると共に、フレームワーク学習問題の解決がますます重要になりつつある。通常フレームワークには正しく利用するために守らなければならない約束事が定められている。フレームワークの趣旨からアプリケーション開発者はフレームワーク内部の実装の詳細な知識無しでそれを正しく利用出来なければならない。そのためにはフレームワーク内部の挙動を抽象化して記述し、正しい利用法を説明する文書が用意されているべきである。

しかしながら現実にはしばしばこうした事柄は文書化されず、例題アプリケーションのコードから正しい利用法を学ばなければならない事例は枚挙い暇が無い [1]。我々は利用法が文書化されていないフレームワークの正しい利用法を例題アプリケーションから学習するための支援手法を追求している。

フレームワークはその複雑性と、制御の反転 (Inversion of Control) と呼ばれる独特な実行形態が正しい利用法を学ぶ際の大きな障害となっている。一般にフレームワーク上に構築されたアプリケーションでは、フレームワーク側のコードが制御の主導権を取る形で実行される。よってアプリケーション固有のコードはフレームワークが定める時機に呼び出される (制御の反転: Inversion of Control [2]) 事を意識して記述される必要がある。一方でアプリケーション固有のコード内からも、しばしばフレームワークの初期化や内部状態の更新のためにフレームワークが公開している API を呼び出す事が求められる。

アプリケーション開発者がここに述べたフレームワークとアプリケーション固有部分の間の相互呼び出しを理解する事は、フレームワークの正しい利用法を学ぶ上で重要である。実際にアプリケーション開発者が自身の書いたコードが呼び出される時機を確認する手法が提案されたり [3]、フレームワーク API の呼び出しに関するプログラミング理解の必要性が指摘されている [4]。

しかしながらフレームワークを利用する趣旨からしても、フレームワークの実装を調べる事は (例えソースコードが公開されている場合でも) 出来る限り避けるべきであると考える。また上で述べたフレームワークとアプリケーション固有部分の間に非常に複雑なメソッド呼び出し関係が形成され、メソッド呼び出しの流れを辿ってその相互作用を理解する事は非常に困難である事が指摘されている [4]。

こうした相互呼び出し関係の理解を支援するために、我々はフレームワーク内部の挙動を**機能**によって抽象的に記述するモデルと、例題アプリケーションの実行トレースからモデルを構築するための解析手法を提案する。我々の機能モデルでは、フレームワークの定める Hot Spots [5] をア

[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)] masa-n@cs.kobe-u.ac.jp
[c)] n-nitta@konan-u.ac.jp
[d)] etsuya@ecc.u-tokyo.ac.jp

プリケーション固有な機能と捉え、上記のやりとりを通じてこれらをフレームワークが予め実装した機能と統合されるものと考える。

フレームワークアプリケーションの挙動を機能モデル上で抽象化する事によって、フレームワークの実装の詳細に立ち入らずにその内部挙動を明らかにする事を目指している。フレームワークとアプリケーション固有部分の間のやりとりがフレームワーク内部に引き起こす複雑な効果を抽象化する事によって、アプリケーション開発者のフレームワーク学習の支援効果が期待される。

本論文では第三者が開発した実用的なフレームワークの誤用例を紹介し、その誤用例に適用する事によって我々の機能モデルの効果を示す。現在我々はアプリケーションの実行トレースを抽象化して可視化する動的解析手法とツールを開発中であり、その現状と将来課題に関しても紹介する。

本論文の残りの内容は以下の通りである。まず第 2 節で、フレームワークアプリケーションに関する概念や用語を説明し、第 3 節でフレームワークの誤用例を紹介する。次に第 4 節で機能モデルと、誤用事例に対して適応した結果を示す。第 5 節で関連研究を、第 6 節で将来課題を議論し、第 7 節で結論を述べる。

2. フレームワークアプリケーション

アプリケーションフレームワークはある特定領域のアプリケーションによって再利用可能な設計と実装を提供するソフトウェア製品である [6]。フレームワークアプリケーションのクラスはフレームワークに属する**フレームワーククラス**とアプケーション開発者が追加した**アプリケーション固有クラス**に分割される。我々はフレームワーククラスが実装するメソッドを**フレームワークメソッド**、アプリケーション固有クラスが実装するメソッドを**アプリケーション固有メソッド**と呼ぶ事にする。

フレームワーククラスの中には特定領域に属する要素を抽象化し、かつその API をアプリケーション開発者に公開した**核心クラス** [7] が含まれている。核心クラスにはアプリケーション開発者がアプリケーション固有の機能として実装すべきメソッドを持つものが含まれている。このようなクラスは *Hot Spots* と呼ばれている。アプリーケーション開発者はこれら Hot Spots を継承し、アプリケーション固有の機能を実装する事によってフレームワークを具体化 (拡張) する。

Hot Spots のメソッドを実装するアプリケーション固有メソッドは、フレームワーククラスのコード内ではフレームワーククラスのメソッドとして参照されている。Pree-emeta:pattern:book:Pree:1995 は参照元のフレームワークメソッドを *template* **メソッド**、参照されているアプリケーション固有メソッドを *hook* **メソッド**と呼んでいる。アプリケーションの実行時には hook メソッドはフレームワークが定める時機に template メソッドから呼び出される、所謂**制御の反転** *(Inversion of Control)*[2] が発生する。

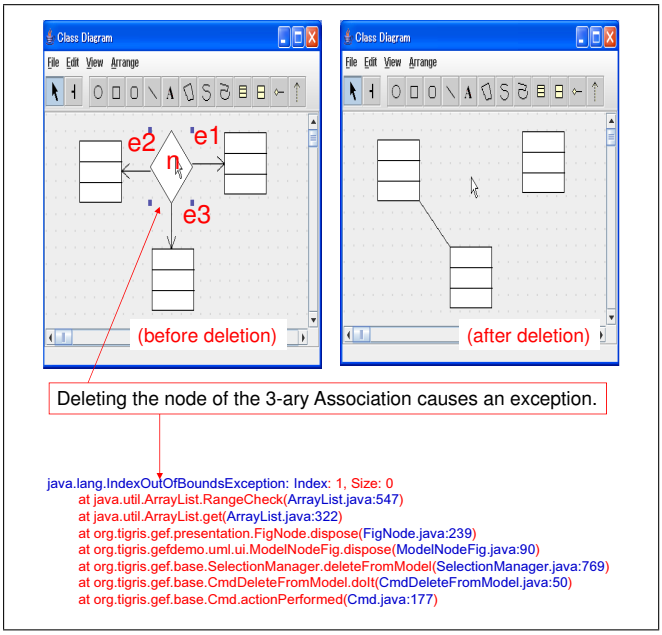

図 1 関連削除時の例外発生
Fig. 1 Exception at Deleting Association Node

実用的なフレームワークのアプリケーションではフレームワークの初期化や、フレームワーク内部の状態変更の引き金を引くために、アプリケーション固有クラスからフレームワークメソッド正しく呼び出す事が求められている。本論文ではこのような呼び出しを制御の反転と対比させて**制御の再反転** *(Re-Inversion of Control)* と呼ぶ事にする。フレームワークを用いたソフトウェア開発の現場ではしばしば制御の再反転に関する約束事が文書化されておらず、既存のアプリケーションのコードからそれらを学ぶ事を余儀無くされている [4], [8]

アプリケーションが正しく実行されるために、アプリケーション開発者は制御の反転に関して整合的なコードを記述する必要がある。一般に hook メソッドを参照する Template メソッドはアプリケーション開発者から隠蔽されており、制御の反転の詳細は明らかにされていない。こうした隠蔽は hook メソッドの呼び出しの試験を困難にする。また、フレームワークの上にアプリケーションを構築するためには単に hot spots を埋める以上の労力が必要であり、クラス間の複雑な協調関係を理解する必要がある。何故なら hot spots を実装するコードは結局のところフレームワークを介して互いに連携しているからである。[1]。

3. フレームワーク誤用例

GEF(Graph Editing Framework)[*1]はグラフデータを編

[*1] http://gef.tigris.org/

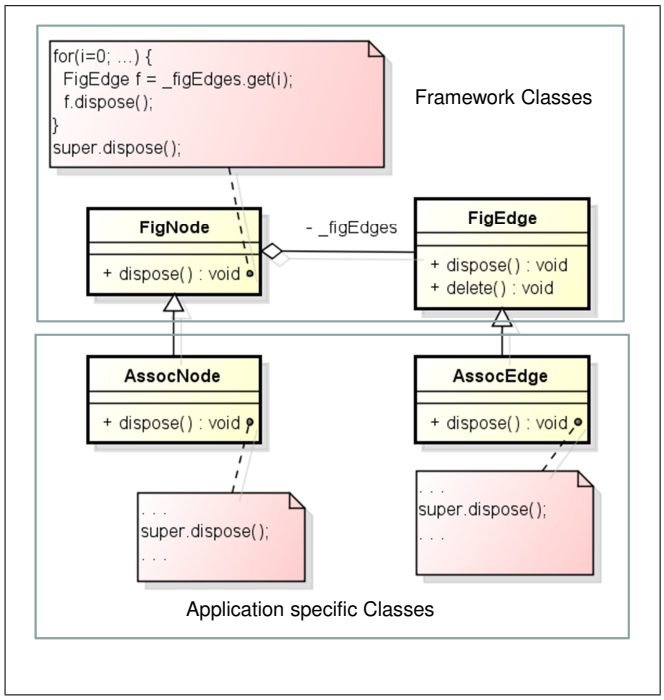

図 2 フレームワークアプリケーションの構成
Fig. 2 Architecture of Example Application

集するアプリケーション用の実用的なアプリケーションフレームワークである。GEF は Java 言語によって記述されたオープンソースソフトウェア製品であり、フレームワーク開発者自身によって作成された UML 編集プログラム[*2]が例題アプリケーションとしてフレームワークに同梱されている。この例題アプリケーションの操作中に発生した例外を図 1 に示す。三つの UML クラスを辺 e1、e2、e3 で結合する 3 項 UML 関連を作成し、その後にその関連を削除しようとした時にこの例外が発生する。より詳細に述べると、利用者が関連を形成する節点 (node) を選択して DELETE キーを打ち込むと、図 1 の右側に示されるような図式と同図下部に示されるようなエラーメッセージが表示される。このエラーメッセージはフレームワーク内部で空のリストから要素を取り出そうとした事を告げており、このリストに関する副作用の発生を示唆している。

この例題アプリケーションのアーキテクチャを図 2 に示す。同図の上部にフレームワークの核心クラス (Hot Spots)FigNode と FigEdge 及びそれらのメソッドの疑似コードを、下部にアプリケーション固有クラス AssocNode と AssocEdge[*3]とそれらのメソッドの疑似コードが配されている。

FigNode と FigEdge はそれぞれ一般的なグラフの節点と辺を表現している。図式中からこれらのクラスのインスタンスを廃棄する際にはフレームワークメソッド ldispose() を呼び出してフレームワークに廃棄を通知する必要がある。

[*2] http://gefdemo.tigris.org/
[*3] 実際のクラス名は非常に長いので、説明を読み易くするために、論文中ではこれらのクラス名を変更している。

一般にグラフの節点を削除する際にはそれにつながる辺も一緒に削除する必要がある。実際に図 2 の左上の疑似コードが示すように、FigNode が定義するメソッド dispose() では自身につながる Edge のインスタンス (疑似コード内では変数 f によって参照されている) に対して dispose() を呼び出している。

アプリケーション固有クラス AssocNode と AssocEdge はそれぞれ FigNode と FigEdge を継承して UML の多項関連の節点と辺を実装している。これらのクラスは多項関連特有の廃棄処理を実装するためにそれぞれがフレームワークメソッド dispose() を上書きしている。AssocNode と AssocEdge のインスタンスはフレームワーク側ではそれぞれ FigNode と FigEdge のインスタンスとして参照されており、アプリケーションの実行時にはこれらのインスタンスに対して dispose() が呼び出される時点で**制御の反転**が実行される。

一方でこれらアプリケーション固有メソッドの内部では、super.dispose(); 命令文によってそれぞれの上位クラスが実装するフレームワークメソッドが呼び出されている。これらのインスタンスはフレームワーク内部で管理されており、アプリケーション固有の廃棄処理を実行すると共にフレームワークにもその廃棄を通知する必要があるために、こうした**制御の再反転**が実装されているのである。

例外発生に関与するメソッド呼び出し木とそこで発生した制御の反転/再反転を図 3 に示す。この呼び出し木は我々が現在実装している動的解析ツールを用いた出力結果に手を加えたものである。赤、青、黄色の矩形や矢印、附記も我々が説明のために手作業で追加したものである。黄色い線や附記はこの呼び出し木に従って関連するソースコードを調査した結果、判明した事柄を示している。

同図中のメソッド呼び出しの冒頭には、記号 F、A、J に続いて角括弧でくくられた数値が一意的に与えられている。上記記号は順にメソッドがフレームワークメソッド、アプリケーション固有メソッド、Java のライブラリメソッド[*4]に分類される事を示している。

メソッドの番号に引き続いてメソッドを定義しているクラス名とメソッドの受け手 (receiver) のクラス名が示されている。例えば以下の表現は Java ライブラリクラス ArrayList インスタンス (#CP に対するメソッド get の呼び出しを表現している。

```
[6]: ArrayList@ArrayList[#CP].get()
```

ここで説明のためにツールの出力結果に対して図 1 の実行例に登場しているオブジェクトの名前が手作業で付与されている。これは図 1 との対応関係を明らかにして説明を分

[*4] ここで取り上げている ArrayList のバイトコードは、実際の解析では Open JDK から生成された同様な機能を持つ別のクラスによって置き換えられている。これは Java のライセンスに定める禁止事項に抵触する可能性を回避するためである

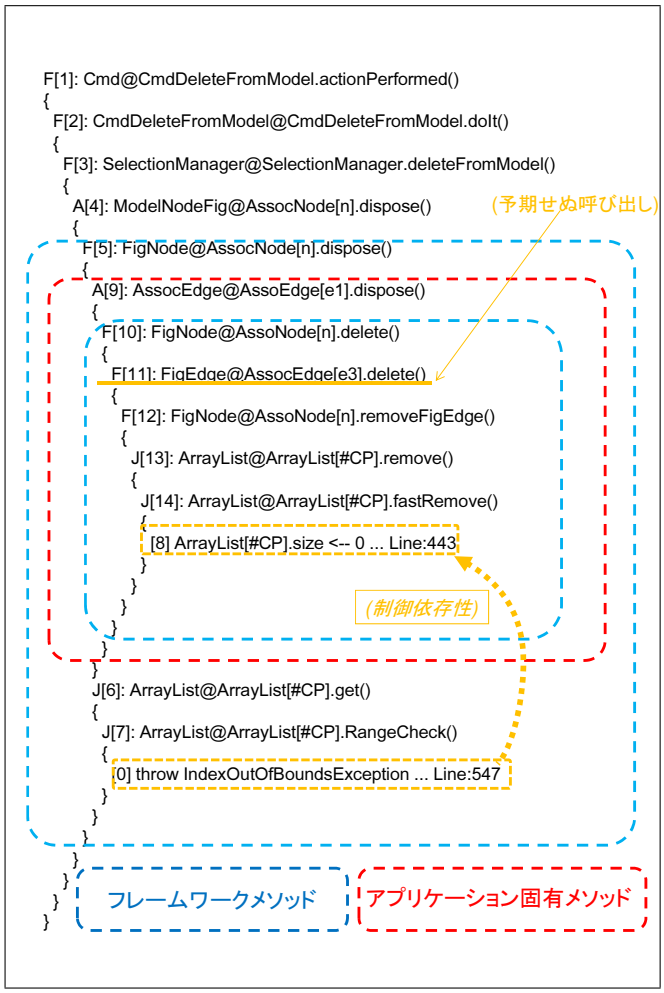

図 3 例外発生時のメソッド呼び出し木
Fig. 3 Invocation Tree at Exception Throwing

かり易くするためである。ここでは多項関係の節点 (node) と辺 (e1 と e3)、及び例外が直接発生したリストオブジェクト#CP に対応する名前が与えられている。

メソッド呼び出しに加えてエラーメッセージで表示された例外の投射 ([0])、リストの要素数を示すインスタンス変数 size への数値 0 の代入 ([8]) が表示されている。このインスタンス変数 (size) はリストから要素を取り出すメソッド呼び出し J[13] の中で参照されている。その値が 0 である、つまりリストが空であるために例外の投射が実行されているのである。こうした制御上の依存関係は例外投射 ([0]) から size への代入 ([8]) への黄色の点線矢印によって明示されている。

例外発生の直接の原因はインスタンス変数 size への代入であるが、その実行を決定付けたのは制御の再反転 (A[9] から F[10]) の呼び出しである。我々がメソッド A[9] のコードをさらに調査した結果、この制御の反転はアプリケーション固有の機能の実現するために必要である事が判明した。最終的にこの不具合はその呼び出し元であるアプリケーション固有メソッド A[4] の実装を修正する事で解決された。以上の調査と修正に関しては我々の過去の研究 [9] で詳細に説明されている。

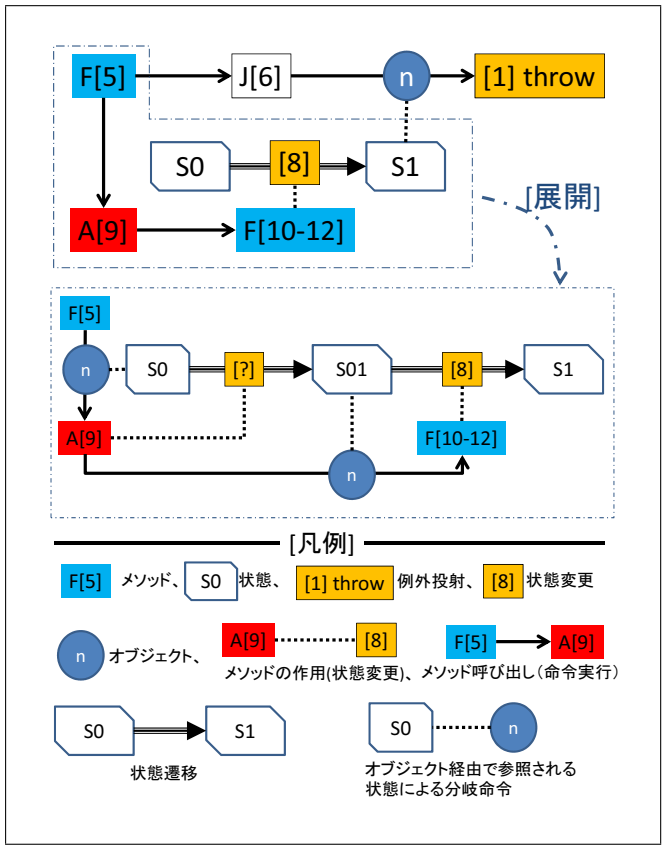

図 4 機能競合 (状態視野)
Fig. 4 Feature Interaction (State View)

4. 提案手法

第 3 節で我々が紹介したようなフレームワークの誤利用例に対して特定し、解決する事はしばしば手間暇を要する作業が必要である。しかも、このような複雑な事例の場合には、フレームワークの内部の処理を調査するために、しばしばアプリッケーション固有部分だけでなく、フレームワークのソースコードもしなければならない事も珍しくない。実際、制御の再反転に関して誤ったメソッド呼び出しが特定出来たとしても、誤りを訂正するコードを書くためにはプログラム理解が必要である事が指摘されている [4]。

しかしながら、本来であればアプリケーション開発者はフレームワークの実装の詳細に立ち入る事無くそれを利用出来るはずである。フレームワークを利用するためにそのソースコードを調査するのは本末転倒に他ならない。こうした二律背反を解決するために、我々はフレームワーク内部の挙動を抽象的な状態の遷移と、より具体的なオブジェクトの参照の視点から捉える機能モデルを提案する。

4.1 機能モデル

我々はモデル化の対象を対話的なフレームワークアプリケーションに限定する。このようなアプリケーションの機能として、アプリケーションの利用者によって実行される

一貫性のある操作 (アプリケーションへの入力) によって、**観察可能な効果**を発生させる能力と定義する。ここで**観察可能な効果**とはアプリケーション利用者が認識出来る出力と、アプリケーション固有部分が関与するメソッド呼び出しやデータの生成・参照を指す。アプリケーション固有クラスやメソッドから参照されるオブジェクトを**観察可能なオブジェクト**と呼ぶ事にする。

我々はアプリケーション開発者は個別のフレームワーク核心クラスに関する知識、特にそれらの API に関する知識を有しているものとする。実際、核心クラスに関する知識が無ければ、そもそもフレームワークの特殊化 (拡張) を行う事が不可能であるため、こうした想定は現実的な側面からも妥当であると考えられる。核心クラスとアプリケーション固有クラスを合わせたものを**観察可能なクラス**、それらのメソッドを**観察可能なメソッド**と呼ぶ事にする。

制御の反転及び再反転それぞれのメソッド呼び出し木の下では結果として観察可能な効果を実現されている。よってこれらの呼び出し木の頂点となる観察可能なメソッドそれ自体が機能を表現しているものと考えられる。これらを**機能メソッド**と呼ぶ事にする。ある機能の実現に関与する観察可能なメソッドはしばしば多数ののぼる。よってこれらを機能メソッドによって代表させる事によって、機能モデルの複雑性の軽減を計る。

実行時に於ける機能の実行は制御の反転と再反転を伴う形で実現され、観察可能な効果とフレームワーク内部の状態変更を引き起こす。前者はアプリケーションコードの実行やアプリケーションの挙動として直接反映されるが、後者は隠蔽されている。機能モデルは後者の内容を、アプリケーション開発者に理解出来る概念、則ち観察可能なオブジェクト、クラスそしてメソッドを用いて表現する事を目的としている。

機能モデルはフレームワークの内部挙動を二つの視点から表現する。一つ目の視点では、フレームワーク部分から参照される複雑なオブジェクトの相互参照関係や、各オブジェクトの状態を単一の状態として抽象化し、その抽象化された状態遷移が観察可能なメソッドの実行と関連付けられている。この関連付けによってメソッド同士の整合的な協調関係 (及び結果としての機能の実現) や、或いは誤利用に由来する競合 (及び結果としての機能実現の失敗) を簡潔に表現出来ると期待される。

二つ目の視点では、オブジェクトの**参照経路** [9] によるメソッド間の協調や競合の実現経緯が表現される。ここでオブジェクトの参照経路とは、あるオブジェクトから他のオブジェクトへ至るインスタンス変数の参照の履歴を意味する。実用的なアプリケーションでは、実際の参照経路にはしばしば観察可能でないオブジェクト、特に Iterator のような一時的に作成されるオブジェクトが含まれている。よってこの視点で表現される参照経路は、こうした観察可能でないオブジェクトの関与を隠蔽する形で抽象化されて呈示されている。

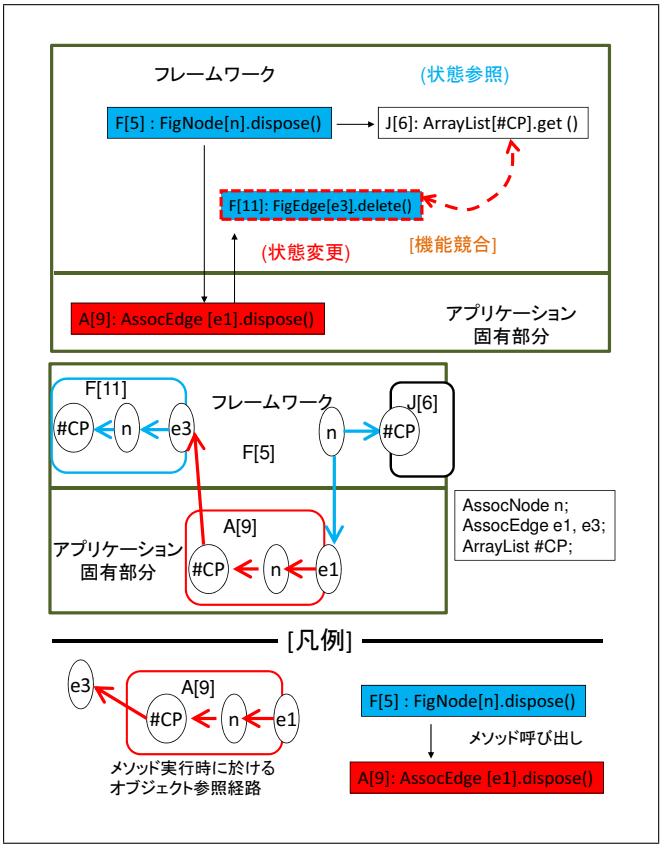

図 5 機能競合 (参照経路視野)
Fig. 5 Feature Interaction (Path View)

参照経路視点で機能を表現する目的は、機能間の協調や競合を実現させるオブジェクトとそれらの互いの関係を明らかにする事にある。また、参照経路を形成するオブジェクトが観察可能なメソッドの受け手や引数、或いは返り値として受け渡されている場合、各メソッドの知識を用いてこれらオブジェクトの果たす役割の理解も期待される。

4.2 適用事例

第 3 節で紹介したフレームワークの誤利用事例に対して我々の手法を適用した結果を図 4 と図 l 5 に示す。前者は抽象状態視点、後者は参照経路視点でそれぞれ多項関係の削除時に発生した観察可能なメソッド同士の競合関係を表現している。図 4 の上段に二つの機能メソッド `F[5]` と `A[9]` が、機能メソッド `F[10]` で始まる制御の再反転によって競合関係に陥る関係が明示されている。具体的には、`A[9]` が `F[10]` を呼び出す事によって、インスタンス変数に値が代入される [8]。さらにライブラリメソッド `J[6]` の内部でその代入結果が観測可能オブジェクト `n`(多項関係の節点) を介して参照され、参照結果を用いた条件分岐命令によって例外投射 [1] が実行されている。

ここで `J[6]` が直接参照しているのは節点オブジェクト `n` ではなく、その内部状態を実装しているリストオブジェ

クトである。実際のところ、このリストオブジェクトこそがメソオッド J[6] の受け手である。しかしながら、この図ではこのリストオブジェクトは、その参照元である節点オブジェクト n によって代表されている。このようなオブジェクトの置き換えは、機能モデルの表示の複雑性を軽減するために行われる。

この機能モデルでは内部状態がこれらの機能メソッドの挙動に与えた影響が表示されていない。よってその影響が隠れている部分 (図中の破線で囲んだ部分) の表示を展開した結果を図中段に示す。ここで初めて制御の反転 (F[5] から A[9] への呼び出し) と制御の再反転 (A[9] から F[10] への呼び出し) の双方が、節点オブジェクト n から取得される内部状態に応じて実行されている事が分かる。特にアプリケーション固有メソッド A[9] は一旦この節点オブジェクトの状態を自分で変更した ([?]) 結果に基づいて F[10] の呼び出しを決定している事が分かる。

上記の観察可能メソッド同士の呼び出しの流れと競合の発生を、それぞれの受け手と共に表示したものを図 5 の上段に示す。ここで機能メソッド F[10] ではなく、その下で呼び出される F[11] が選択されているのは、その受け手である多項関係の辺 e3 の登場に保守作業者が注目したからである。そもその、この一連の処理は多項関係の節点 n の廃棄 dispose() に伴う辺 le1 の廃棄によって開始されている。その処理の延長上に別の辺 e3 の削除メソッド delete() が呼び出される事を保守作業者は予想していなかった。よって F[10] の代わりに F[11] を選択したのである。

それではこの意外な辺は他のオブジェクト (n と e1) とどのような関係を辿って登場したのだろうか？中段に表示されている機能モデルがこの疑問に応えている。このモデル表示から、この意外な辺は今まさに廃棄されようとしている e1 から、それが繋ぐ節点 n(とそれが抱えているリストオブジェクト (#CP)) を経由して取得されているのである。図 4 の中段に示された A[9] の挙動から、この機能メソッドの判断によってこの関連性が辿られている事が分かる。

以上の結果から、観察可能なメソッド同士の競合はこの機能メソッド A[9] が自分で変更した節点 n の状態に従って実行された制御の再反転が原因である事が判明した。この制御の再反転を行うという判断が問題解決の鍵となる事が示唆されたのである。

以降の調査は実際にアプリケーション固有メソッドである A[9] のソースコードの読解が必要である。詳細は我々の過去の研究 [9] に譲るが、調査の結果この判断はアプリケーション固有の処理の実現上必要なものである事が判明したた。結局 A[9] (AssocNode のメソッド dispose()) が F[5] 経由で呼び出される場合には F[10] を呼び出さないようにするような変更を加える事によって、この誤利用問題は実際に解決された。

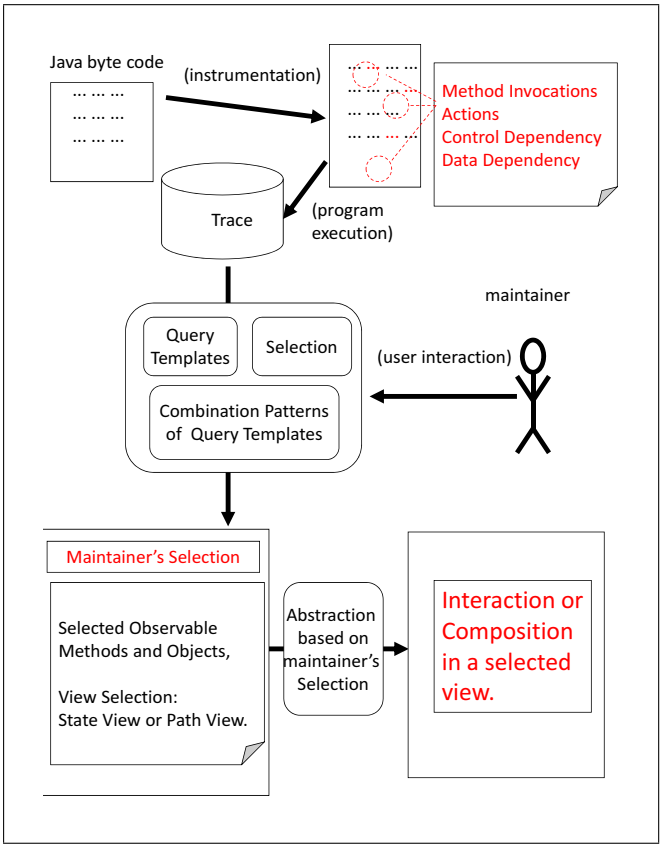

図 6　動的解析処理の概要
Fig. 6　Overview of Dynamic Analysis

4.3　動的解析による実現に向けて

第 4.2 節で説明したような機能モデルを動的解析によって構築するためには、トレースに含まれているのが単なるメソッド呼び出し関係だけでは明らかに不十分である。トレース中にはメソッドが実行する命令や、命令の実行に関与するオブジェクトや値、そしてこれらのデータの依存関係や、制御の依存関係も含まれていなければならない。さらにデータの依存関係は、オブジェクトのインスタンス変数や配列[*5]を介した依存性を包含するものでなければならない。また、メソッド呼び出しやインスタンス変数の操作に関与するオブジェクトも特定される必要がある。

このように我々が必要とするデータの一部、あるいは大部分を含むトレース生成手法は既に研究されている。(例えば [10], [11]) しかしながら、我々の機能モデルを構築するためには、それに加えてさらにトレースに対する検索やモデルの構成要素であるメソッドやオブジェウトの指定も可能である事が求められる。実際のところ、我々は解析作業者が問い合わせや選択を効率的に実施出来るような動的解析ツールと、それを用いた作業のパターンを開発中である。

現在我々は必要なデータ種を生成するバイトコードの可測化 (byte code instrumentation) 処理と基本的な問い合わせ機能の実装を完了し、現在では問い合わせのパターン化

[*5] 例えばここで登場するリストは配列を用いて実装されている。

と、パターン化された問い合わせの組み合わせ機能を実装中である。現状では図 4 図 5 に示されるような表示の部分は実装されておらず、両図の表示は現在の実装で生成されたテキスト形式のデータから手動で作成されている。我々の想定する解析作業とツールの処理の概要を図 6 に示す。

5. 関連研究

5.1 フレームワークアプリケーション学習支援

Tyler と Soundarajan はフレームワークアプリケーションによって隠蔽された、制御の反転の時機を試験する枠組みを提案している [3]。Shull 等 [1] は多数のプロジェクトに対して幾つかの既存の手法を適用した実証実験を通じてフレームワーク学習に関する仮説を検証している。

Monperrus 等 [4] は制御の再反転を実装する際のメソッド呼び出しのパターンの誤りを統計的に特定する手法を提案している。彼等の手法はメソッド呼び出しの字面上の解析に基いているが、誤りを修正する作業に於いてはより深いプログラム理解の必要性にも言及している。Heydarnoori 等 [8] は同一のフレームワークに対して二つの例題アプリケーションを実行し、そのトレースから指定された機能を実現するために必要な制御の再反転を構成するフレーワークメソッド呼び出しを抽出する手法を提案している。こうした抽出に際して彼等はメソッド呼び出しに於けるメソッドの受け手、引数、返り値として登場するオブジェクトの参照関係を利用している。

5.2 機能に関連する動的解析

動的解析は可測化 (instrumentation)[12] されたプログラムコードを実行し、その実行履歴を記録したトレースからソフトウェアの挙動のある側面を解析する手法である。動的解析は機能捜索 (feature location) の一環として機能実行 (feature at runtime) に関与するクラスやメソッドを特定するためにしばしば利用される [13], [14], [15], [16], [17], [18], [19]。こうした対応関係の解析の主な対象はメソッド呼び出しである。一般にトレースには保守作業者が直接扱えないほど膨大な回数のメソッド呼び出しが含まれているため、呈示する情報量を軽減するための何らかの手法が鍵となる。

作業者とツール間の対話による情報の選択や、抽象化された表示、つまりユーザインタフェイスは情報量軽減を実現する重要な要因の一つである。Richner と Ducasse は作業者による問い合わせによって複雑なオブジェクト間協調を段階的に調査するための環境を構築した [13]。Röthlisberger 等は機能を試験するためのテストケースを活用して複雑なメソッド呼び出し構造を機能木 (feature tree) として抽象化し、ソースコードと対応付けて閲覧する環境を構築し、被験者を用いた実験によってその有効性を示している [14]。Cantal 等 [20] はクラスのパッケージ名やクラスの一般性 (Utility クラスか否か) に関する利用者や知識や、クラスの呼び出しパターンによるトレースのふるいわけを利用者が繰り返す手法を提案している。

メソッド呼び出しに関する情報量を自動的に削減するために、メソッド呼び出しや依存関係の解析を基本として様々な技術との組み合わせが試みられている。Salar 等 [15] はある特定のクラスに対する他のクラスからのメソッド呼び出し系列同士の類似性を利用して、そのクラスの利用シナリリオを特定する手法を提案している。Eaddy 等 [16] はメソッド呼び出し関係やクラス間の参照関係、及びコメントや識別子に対する情報検索技術との組み合わせ手法を提案している。Greevy と Ducasse [17] は利用者が GUI を通じて起動し (user-initiated)、かつその結果を観測可能な (user-observable) 機能に対するメソッドやクラスの寄与を、メソッドの呼び出し回数やメソッドやクラスの参照回数を利用して特定する手法を提案している。Eisenbarth 等 [21] は関数やメソッドのようなプログラム要素と機能を Formal Concept Analysis のオブジェクトと属性と見做して機能と関数群を対応付ける手法を提案している。

実用的なプログラムで実行される機能同士はシステムの複雑な内部状態によって互いに関連付けられている事が多い。ソフトウェア開発や保守作業の中でこのような関連をを特定するためにはメソッド呼び出しに加えてデータや制御の依存関係の解析も必要となる。Salah と Mancoridis [22] はオブジェクトの生成に関する依存関係を利用して機能間の依存関係を解析した。Lienhard はオブジェクトの生成に加えて実行時の参照構造も利用した Dynamic Object Flow Analysis [23] を提案している。Wang と Roychoudhury [18] はデータの依存性を利用してトレースを段階 (phase) に分割する手法を提案している。

機能同士の重要な関係の一つとして**機能競合**が挙げられる。機能競合は個別には問題無く実行される複数の機能が組み合わされる事によって予期せぬ結果を生む現象を意味している。本来は電話システムやウェブ上のアプリケーションシステム (例えば [24], [25]) の分野で研究されてきた。我々の過去の研究 [9] ではフレームワークの誤利用によって発生した副作用の事例を機能競合として形式化し、動的解析によって特定する手法が提案されている。

6. 議論

実用的な解析手法を実現する上の課題として、解析処理の効率性、問い合わせ処理のユーザインタフェイス、さらにアプリケーションの挙動と観測可能なオブジェクトの対応付け (例えば多項関連を形成する図とそれを実現するオブジェクトの対応付け) の自動化が挙げられる。効率性の観点からすると、トレースは当然軽量化される事が望ましいが、問い合わせや選択に必要な情報量を保つ事も求められる。研究の現段階では、解析処理の効率性よりも要求されるトレースのデータ種の豊富さの実現に重点を置いて

いる。

機能モデルの表示と一体化した問い合わせのユーザインタフェイスを現在設計中である。ユーザインタフェイスの有効性を検証するためには、被験者を用いた実証実験が必要であると考えている。オブジェクトの対応付けに関しては、Java ライブラリの描画 API から描画データを供給しているオブジェクトを特定する手法を検討中である。

7. おわりに

本論文で我々はフレームワーク利用の学習支援を目的とした機能モデルと、動的解析によるその実現に向けた我々の取り組みを説明した。我々の機能モデルは、フレームワークアプリケーション固有の実行形態である制御の反転と再反転の組み合わせとして機能の実現を表現している点に特徴がある。通常では理解や解決が困難であるような複雑なフレームワークの (実在する) 誤利用事例に我々の手法を適用する事によってその有用性を議論した。さらに将来課題として、我々の機能モデル構築する動的解析手法とツールの現状を将来課題に関して説明した。

謝辞 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)、基盤 (B) (No.23300009)]、及び関西エネルギー・リサイクル科学研究振興財団の助成を受けています。

参考文献

[1] 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).

[2] Sparks, S., Benner, K. and Faris, C.: Managing Object-Oriented Framework Reuse, *IEEE Computer*, Vol. 29, No. 9, pp. 52–61 (1996).

[3] Tyler, B. and Soundarajan, N.: Testing Framework Components, *Component-Based Software Engineering*, LNCS 3054, Springer, pp. l38–145 (2004).

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

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

[6] Johnson, R. E. and Foote, B.: Designing Reusable Classes, *Journal of Object-Oriented Programming* (1988).

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

[8] Heydarnoori, A., Czarnecki, K. and Bartolomei, T. T.: Supporting Framework Use via Automatically Extracted Concept-Implementation Templates, *ECOOP*, LNCS 5653, Springer-Verlag, pp. 344–368 (2009).

[9] 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).

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

[11] 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).

[12] Reiss, S. P. and Renieris, M.: Languages for Dynamic Instrumentation, *International Workshop on Dynamic Analysis* (2003).

[13] Richner, T. and Stéphane Ducasse: Using Dynamic Information for the Iterative Recovery of Collaborations and Role, *International Conference on Software Maintenance*, IEEE, pp. 34–43 (2002).

[14] David Röthlisberger and Orla Greevy and Oscar Nierstrasz: Feature Driven Browsing, *International Conference on Dynamic Languages*, ACM, pp. 79–100 (2007).

[15] Salah, M., Denton, T., Mancoridis, S., Shokoufandeh, A. and Vokolos, F. I.: *Scenariographer*: A Tool for Reverse Engineering Class Usage Scenarios from Method Invocation Sequences, *International Conference on Software Maintenance*, IEEE, pp. 155–164 (2005).

[16] Eaddy, M., Aho, A. V., Antoniol, G. and Yann-Gaël Guéhéneuc: Cerberus: Tracing Requirements to Source Code Using Infomation Retrieval, Dynamic Analysis, and Program Analysis, IEEE, pp. 53–62 (2009).

[17] Ducasse, O. G. S.: Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach, *European Conference on Software Maintenance and Reengineering*, IEEE, pp. 314–323.

[18] Wang, T. and Roychoudhury, A.: Hierarchical Dynamic Slicing, *International Symposium on Software Testing and Analysis*, ACM, pp. 228–238 (2007).

[19] Eisenbarth, T., Koschke, R. and Simon, D.: Feature-Driven Program Understanding Using Concept Analysis of Execution Traces, *International Workshop on Program Comprehension*, IEEE, pp. 300–309 (2001).

[20] de Sousa, F. C., Mendonca, N. C., Uchitel, S. and Kramer, J.: Detecting Implied Scenarios from Execution Traces, *Working Conference on Reverse Engineering*, IEEE, pp. 50–59 (2007).

[21] Eisenbarth, T., Koschke, R. and Simon, D.: Feature-Driven Program Understanding Using Concept Analysis of Execution Traces, *International Workshop on Program Comprehension*, IEEE, pp. 300–309 (2001).

[22] 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).

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

[24] Nakamura, M., Igaki, H. and ichi Matumoto, K.: Feature Interactions in Inegrated Services of Networked Home Appliances, *Feature Interactions in Telecommunications and Software Systems*, pp. 236–251 (2005).

[25] Wilson, M., Kolberg, M. and Magill, E. H.: Considering Side Effects in Service Interactions in Home Automation - an Online Approach, *ICFI*, pp. 172–187 (2007).