情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-SE-174 No.6
2011/11/2

デルタ抽出を用いたアプリケーションフレームワークの利用例抽出手法

手 塚 裕 輔[†1] 新 田 直 也[†1]

近年アプリケーションフレームワークは, 拡張可能なアプリケーションの骨組みとしてさまざまな分野で広く用いられ成果をあげている．しかしながら, 個々のアプリケーションを実装する上でアプリケーションフレームワークをどのように用いればよいかは, 開発者にとって自明でない場合が多く, 実装前の調査に多くの時間を要したり, 実装後の予期せぬ不具合発生の原因となっている．本研究では, その中でも特にプラグインポイントに代表されるようなアプリケーションフレームワークの拡張に伴って暗黙的に必要とされるメソッド呼び出しに着目し, アプリケーションフレームワークの単一のサンプルアプリケーションの実行履歴からそれらの呼び出しを抽出することで開発者を支援する手法を提案する．ここで実行履歴の解析にあたっては, 本研究室で提案しているデルタ抽出を用いる．本稿では, 2 つのアプリケーションフレームワークを対象にデルタ抽出を適用し, 利用例の抽出を行った事例研究について紹介する．

A Method to Extract an Application Framework Usage based on Delta Mining

Yusuke Tezuka[†1] and Naoya Nitta[†1]

Recently, application frameworks are widely used as skeletons of applications in various domains. However, it is not clear for application developpers how to reuse an application framework to build their own applications, and they are often needed to spend much time investing the application framework for their implementation tasks or face with unexpected defects after the implementation tasks. In this research, we focus on implicitly needed method calls along with an extention of an application framework such as plugin points, and present a method to extract such calls from an execution trace of a single sample application of the application framework to support application developpers. For the analysis of the execution trace, we use delta mining method which we have proposed in the previous research. In this paper, we show case studies to extract framework usages for two application frameworks using delta mining method.

1. はじめに

近年, さまざまな分野のソフトウェア開発においてアプリケーションフレームワークの利用が一般化しつつある．アプリケーションフレームワーク[1)](以下, 本稿ではフレームワークとよぶ) は, 主にオブジェクト指向技術に基づいて, 特定のドメイン内で繰り返し出現するオブジェクト間の協調やアプリケーション全体の制御構造などを抽象化し再利用可能にしたクラス群であり, アプリケーションの中核となる設計だけでなく, その実装も柔軟に再利用することができるという特徴を持つ．そのため, アプリケーションの開発において適切なフレームワークを選択すれば, 設計や実装に要する工数を効果的に削減することができると期待される．

しかしながら, 個々のアプリケーションを実装する上でフレームワークをどのように再利用すればよいかは, アプリケーション開発者にとって自明でないことが多く, ある程度ドキュメントが整備されているオープンソースフレームワークにおいても問題となることが指摘されている[2)]．一般にフレームワーク側のクラス内にはアプリケーション側で変更可能なホットスポット[3)] が抽象メソッドとして用意されており, アプリケーション開発者は, このような抽象メソッドをアプリケーション側で作成した子クラスでオーバーライドすることによってアプリケーション固有の振る舞いを実装する．このとき, アプリケーション側で実装したメソッドはフレームワーク側から呼び出されることになる．このような制御の仕組みを一般に**制御の反転**とよぶ.

ホットスポットはフレームワークが提供している主要な機能であるため, その情報をアプリケーション開発者が知ることは比較的容易であるが, フレームワークを正しく利用するためにはホットスポットの拡張に加えて, フレームワーク側のいくつかのメソッドを正しい場所で正しい順序で呼び出す必要がある場合が多く, アプリケーション開発者の混乱を招く大きな要因となっている[5),6)]．とりわけ**プラグインポイント**[6)] は, 制御の反転を起こさせるために呼び出さなければならないフレームワーク側のメソッドであり, アプリケーション側で拡張した機能をフレームワークに組み込むために必要となるものである．本研究では, Java プログラムを対象に, このような制御の反転を起こさせるためにアプリケーション側に記述

†1 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University

© 2011 Information Processing Society of Japan

しなければならないコードを, サンプルアプリケーションから抽出する手法の構築を目指す. 具体的には, サンプルアプリケーションを実行させて制御の反転を生じさせ, そこに至る実行履歴を解析して, その制御の反転を起こさせるために必要なアプリケーション側からフレームワーク側への呼び出しをフレームワークの利用例として抽出することを考える.

一般に制御の反転が発生する直前には, その時点の呼び出しスタック中に存在している参照を経由した呼び出しが行われたはずである. そのため, 事前にそれらの参照が生成されていることが制御の反転の発生には必要となるが, さらにそれらの参照が生成されるためにはそれ以前に別の複数の参照が生成されている必要があるといった形で, 参照の生成は実行時に依存する. 本研究では, サンプルアプリケーションの実行履歴の中で参照生成の依存関係を実行とは逆向きに辿ることによって, 制御の反転の発生に寄与しているアプリケーション側からフレームワーク側への呼び出しをその実行履歴中から探し出すアプローチを採る (詳細については 4 節を参照).

本研究室では, 実行履歴中の参照生成の依存関係を効率よく遡る手法としてデルタ抽出[11) を提案している. 本稿ではデルタ抽出を用いて, いくつかのフレームワークを対象に, サンプルアプリケーションの実行時に発生する制御の反転からフレームワークの利用例を抽出することを試みた. その結果, ドキュメントの情報を使わなくても効率良く適切な利用例を抽出できることがわかった.

2. フレームワークの利用例抽出

フレームワークの利用例には様々な種類のものがあるが 4), 本稿では, プログラムを実行した時に特定の制御の反転を発生させるためにアプリケーション側に記述しなければならないコードを対象とする. たとえば, **図 1** のプログラム (クラス図を**図 2** に示す) において, クラス A, B, C をフレームワーク側, クラス SubA, SubC をアプリケーション側のクラスと考えたときに, このプログラムを実行して制御の反転を発生させるためには, アプリケーション側に 16∼18 行の 3 行を書かなければならない. その理由について以下に説明する.

まずこのプログラムでクラス SubA の start() メソッドを実行すると, 制御の反転によりクラス B の exec() メソッドからクラス SubC の exec() メソッドが呼ばれる. このプログラムの実行においてこの制御の反転が発生するための条件を考える. まずこの実行においては, クラス B の exec() メソッドが呼び出されなければこの制御の反転は起こらない. さらに, B の exec() メソッドを呼び出すためには 18 行の A.exec() の呼び出しが必要である. また SubC の exec() が正しく呼び出されるためには, クラス B のインスタンスからクラス SubC のインスタンスへの参照が事前に生成されている必要がある. この参照が生成されるためには 17 行で SubC をインスタンス化し, かつそのインスタンスを引数として A.attach() メソッドを呼び出さなければならない. また, A.attach() を呼び出したときにその内部で B.attach() が呼び出され, クラス B のインスタンスからクラス SubC のインスタンスへの参照が生成されるが, その際に事前に A のインスタンスから B のインスタンスへの参照が生成されていなければ B.attach() の呼び出しで実行時エラーが発生する. この参照は 16 行の実行によって生成されるため A.init() の呼び出しも必要となる. このように制御の反転を起こすためには, 事前に呼び出さなければならないフレームワーク側のメソッドが存在することがわかる.

このサンプルプログラムは説明のために簡略化しているが, 実際のフレームワークでははるかに複雑な実行時の依存関係が存在し得る. そのため, 一般にこのような利用例を人手で抽出するには多大な時間を要すると考えられる.

```
 1: class A {                          21: class B {
 2:    B b = null;                     22:    C c = new C();
 3:    void init(){                    23:    void attach(C c){
 4:       b = new B();                 24:       this.c = c;
 5:    }                               25:    }
 6:    void attach(C c){               26:    void exec(){
 7:       b.attach(c);                 27:       c.exec();
 8:    }                               28:    }
 9:    void exec(){                    29: }
10:       b.exec();                    30:
11:    }                               31: class C {
12: }                                  32:    void exec(){
13:                                    33:    }
14: class SubA extends A {             34: }
15:    void start(){                   35:
16:       init();                      36: class SubC extends C {
17:       attach(new SubC());          37:    void exec(){
18:       exec();                      38:    }
19:    }                               39: }
20: }                                  40:
```

図 1 フレームワークの利用例の例 (ソースコード)
Fig. 1 An example of framework usage (source codes)

© 2011 Information Processing Society of Japan

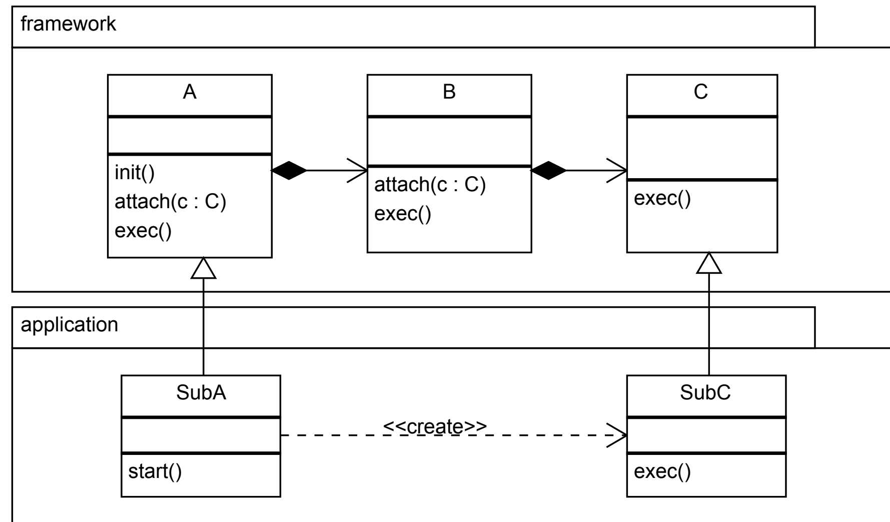

図 2 フレームワークの利用例の例 (クラス図)
Fig. 2 An example of framework usage (class diagram)

```
 1: class Main {
 2:    A a = new A();
 3:    void main() {
 4:       a.m();
 5:    }
 6: }
 7:
 8: class A {
 9:    B b = new B();
10:    D d = new D();
11:    void m() {
12:       d.passB(b);
13:    }
14: }
15:
16: class D {
17:    E e = new E();
18:    void passB(B b) {
19:       e.setC(b.getC());
20:    }
21: }
22:
23: class B {
24:    C c = new C();
25:    C getC() {
26:       return c;
27:    }
28: }
29:
30: class E {
31:    C c = null;
32:    void setC(C c) {
33:       this.c = c;
34:    }
35: }
36:
```

図 3 デルタのサンプルプログラム
Fig. 3 An example program of delta

3. デルタ抽出

本研究では, フレームワークの利用例抽出に我々研究室で提案しているデルタ抽出と呼ばれる実行時解析技術を適用することを考えている. デルタ抽出の概要について以下に説明を行う (定義の詳細については文献 11) を参照されたい). 一般にある参照の生成には, それ以前に生成された他の参照が関与し得る. たとえば**図 3** に示したプログラムでは, 33 行の実行に伴って, クラス E のフィールド変数 c によるクラス C のインスタンス (o とおく) への新たな参照が生成されるが, その生成には 32, 19, 26, 24 行に出現する同じ o への参照 (エイリアス) だけでなく, たとえばクラス B のインスタンスを参照している 12 行の b も間接的に関与している. なぜならば, この b がクラス B の別のインスタンスを参照していた場合, 19 行の getC() メソッドによって取得される C のインスタンスも o とは別のものに変わり得るからである. そこで本研究では, プログラムの実行中に出現する特定の構造 (デルタと呼ぶ) を抽出することによって, 着目している参照の生成に関与している他の参照を網羅的に探索することを考える. デルタの基本的な考え方は以下の通りである. 図 3 に示したプログラムを例に考える. このプログラムを実行すると, **図 4** のオブジェクト図で示したようなオブジェクトとその間のフィールドを介した参照が生成される. またこの実行例のシーケンス図を**図 5** に示す. 以下では説明の簡単のため, 各クラスのインスタンスを図 4 中に示したオブジェクト名で呼ぶ. ここでは, オブジェクト e がオブジェクト c をフィールド c で参照するようになった経緯について考える. まず, e が c を参照するためには, c が e に何らかの手段で渡されなければならない. そのためには, e と c の両方を “知っている” オブジェクトが必要である. このプログラムでは, オブジェクト a がそれに相当する. e から c への参照 r は, a が c を e に “紹介” することによって生成されたとみなすことができる. このとき, a, c, e を結んで得られる構造を r の**デルタ**と呼び, a が c を e に紹介する際に行われたメソッド m() の実行を r の**コーディネータ**と呼ぶ. また, r から r のデルタを求めることを**デルタ抽出**と呼ぶ. デルタは任意の参照 r に対してちょうど 1 つ定義することができる. 直観的にある実行例において生成された参照 r のデルタは, その実行例において r が生成されるために必要な事前条件を表す.

© 2011 Information Processing Society of Japan

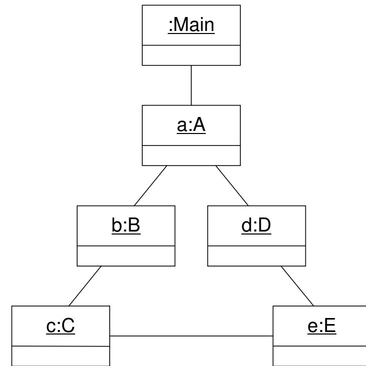

図 4 サンプルプログラムのオブジェクト図
Fig. 4 An object diagram of the sample program

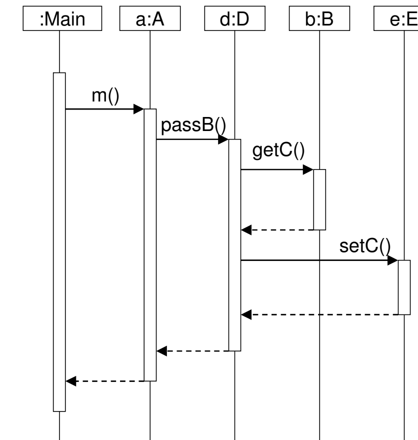

図 5 サンプルプログラムのシーケンス図
Fig. 5 A sequence diagram of the sample program

4. デルタ抽出に基づくフレームワークの利用例抽出

2 節では, サンプルアプリケーションの実行時に生じる制御の反転に着目し, それを起こすためにアプリケーション側に記述しなければならないコードをサンプルアプリケーション中から抽出する問題を, フレームワークの利用例抽出として定義した. 多くの場合制御の反転が発生する際には, フレームワーク側クラス内で宣言された変数によるアプリケーション側クラスのインスタンスへの参照が使われる. そのため, 事前にその参照が生成されていることが制御の反転が生じるための必要条件となる. このような参照を制御の反転に直接関与する参照と呼ぶ. さらに制御の反転に直接関与する参照の生成には, その参照が生成される前に生成された複数の別の参照が関与している場合が多い. したがって, 制御の反転に直接関与している参照だけでなく, その参照が生成されるために間接的に関与している参照も正しく生成されていることが, 制御の反転が起きるための事前条件となる. ところでそれらの参照の一部が, その実行履歴中のアプリケーション側からフレームワーク側への呼び出しを起点として生成されている場合がある. その場合, それらの呼び出しを行うコードがサンプルアプリケーション内に記述されていなければ, その実行例の中で制御の反転が生じることはない. そこで本稿ではフレームワークの利用例抽出を, サンプルアプリケーションの実行履歴を対象として, 制御の反転に直接的または間接的に関与している参照の生成を引き起こすアプリケーション側からフレームワーク側への呼び出しを抽出する問題として捉える.

3 節では, 実行履歴の中からデルタという構造を抽出することによって, 参照間の実行時の依存関係を捉える手法について説明した. そこで本節では, フレームワークの利用例抽出にデルタ抽出を適用することを考える. あるフレームワークのサンプルコードが与えられたとき, その中の利用例は基本的に以下のような手順にしたがって抽出することができる. まず, サンプルコードのある実行例の中で発生した制御の反転について, その制御の反転に至る呼び出しスタック (以降ではこれを最終スタックと呼ぶ) 中の呼び出しで用いられたすべての参照に対してデルタを抽出する. 次に, 抽出されたデルタを構成するすべての参照 (デルタ内参照) とコーディネータに至る呼び出しスタック中で利用されたすべての参照 (呼び出しスタック内参照) に対して再帰的にデルタを抽出していく作業を繰り返すことによって, その実行例の中で制御の反転に直接的または間接的に関与した参照を網羅的に抽出する. この作業は新しいデルタが抽出されなくなるまで繰り返す. 最後に, 抽出されたすべてのデルタのコーディネータへの呼び出しスタックを参照し, その中にアプリケーション側からフレームワーク側への呼び出しが存在するか否かを調べて, 存在した場合その呼び出しを抽出しその実行例におけるフレームワークの利用例とする.

© 2011 Information Processing Society of Japan

フレームワークの利用例の抽出は基本的には上記のような手順によって進めることができるが, 利用例抽出の目的や状況に応じて, 抽出されたデルタや利用例の情報を元にデルタ抽出を実行する参照を制限したり, デルタの抽出を行う順序を制御することによって, 抽出されるデルタの総数を抑制することも考えられる. たとえば以下のような場合が想定される.

- **利用例抽出作業の効率化:** たとえばデバッガを用いて人手でデルタを抽出する場合, フレームワークの利用例の抽出過程の中で一時的に抽出されるデルタの数が爆発し, 人手で実行可能な範囲内に総作業量が収まらなくなる可能性がある. そのため, 結果に影響を与えない範囲で一部の参照を追跡対象から外すことによって抽出数の爆発を抑制し, 総作業量を抑えることが考えられる. 明らかにフレームワークの利用例の抽出に影響を与えない参照としては, たとえばフレームワークを利用する場合は必ずフレームワーク内で生成される参照などが挙げられる.
- **抽出した利用例の汎用性向上:** デルタの抽出数を抑制する他の理由として, そもそも結果として抽出されるフレームワーク利用例のサイズを抑えたい場合も考えられる. たとえば, フレームワークのサンプルアプリケーションが固有の機能を多く含んでいて, 作業者が実装したいアプリケーションの機能と大きく異なる場合, たとえ同じ制御の反転に着目しているとしても, 抽出された利用例がそのままの形で作業者のアプリケーションの中で利用できない可能性が高い. これはサンプルアプリケーションから抽出される利用例が, そのサンプルアプリケーション固有の実装の影響を受けてしまうためである. いっぽう, ある制御の反転のプラグインポイントを抽出することができれば, 作業者がどんなアプリケーションを実装したい場合でも, その制御の反転を使う限りにおいて安全に利用することができると考えられるが, これはその制御の反転を起こすために任意のアプリケーションで必要とされるような利用例を抽出することに相当する. しかしながら我々の手法は, 単一のサンプルアプリケーションのある実行例のみから利用例を抽出することを試みるため, 一般にプラグインポイントより大きな結果を利用例として抽出してしまう. そこで, 一部の参照を追跡対象から外すことによって, 最終的に得られる利用例が抽出したいプラグインポイントに近づくようにすることが考えられる. そのために追跡対象から外すことができる参照としては, たとえば作業者が実装したいアプリケーションにおいて使用されないことがわかっているオブジェクトに関わる参照などが挙げられる.

デルタは, フレームワークの利用例を実行履歴の中から抽出する作業における一種の中間生成物であるが, 上記のように作業者がサンプルアプリケーションの実行履歴を理解しながら, 一部の参照を追跡対象から外す過程においては非常に有用であると考えられる.

5. 事例研究

提案手法の有効性を検証するため事例研究を行った. 対象としたフレームワークは統合開発環境である eclipse[7] の SWT(Standard Widget Toolkit) と 3D ゲームエンジンの jMonkeyEngine[8] の 2 つである. それぞれの事例について, 手法の有効性と実効性の 2 点の評価を行うため, 得られたフレームワークの利用例が妥当なものであるかどうかの検証と, デルタ抽出を行った結果得られた参照の総数, 及びその中で再帰的にデルタ抽出を実行した参照数 (着目参照数) のデータを収集した. なお, 本事例研究では著者の一人がデバッグ機能等を用いることで手動でデルタ抽出を行い, 着目する参照については抽出過程の中で適宜判断した.

5.1 SWT

eclipse 内部には SWT を含め様々なサブシステムが存在するが, そのうちの一つである jFace が SWT をフレームワークとして利用している. 本事例では, eclipse 全体をサンプルアプリケーションと考えて, jFace が SWT をどのように利用しているかを抽出した. 対象とする制御の反転は org.eclipse.swt.widgets パッケージの EventTable クラスから org.eclipse.jface.action パッケージの ActionContributionItem$6 クラスを呼び出すものとし, その制御の反転が発生した時点でブレークポイントを用いて実行を中断し, その時点のスタックを最終スタックとした. 最終スタックに現れた参照から着目する参照を選び 4 節で述べた手法に基づいて再帰的にデルタ抽出を行った結果, 抽出されたデルタは 7 個, フレームワークの利用例は 3 例見つかった (表 1 参照). また, デルタ抽出の過程で確認された参照の総数は 41 だった. (詳細を表 2, 3 に示す.)

5.2 jMonkeyEngine

jMonkeyEngine[8](以下, jME と略す.) はオープンソースのゲームエンジンである. 実装言語は Java で, 規模は約 31.5 万行である. 全体のアーキテクチャはフレームワーク形式を採用している. jME は物理演算の機能を含まないメインサブシステムの上位に物理演算サブシステムをのせた 2 層構成になっていて, 物理演算サブシステムは自由に着脱可能である. 本事例では物理演算サブシステムを含めた全体を一つのフレームワークと考えて検証を行う. 解析の対象とするサンプルアプリケーションは物理演算サブシステム内のチュートリアルプログラム (com.jmetest.physicstut.Lesson8) とする. このプログラムは 1 つのクラスから成り, 規模は約 150 行, アプリケーション側からフレームワーク側への呼び出しは全体

© 2011 Information Processing Society of Japan

表 1 抽出された利用例
Table 1 Extracted usages

	利用例
U1-1	ToolItem.addListener() の呼び出し
U1-2	new Listener() の実行
U1-3	ToolItem.ToolItem() の呼び出し
U2-1	InputHandler.addAction() の呼び出し
U2-2	PhysicsNode.getCollisionEventHandler() の呼び出し
U2-3	DynamicPhysicsNodeImpl.generatePhysicsGeometry() の呼び出し

で約 55 箇所存在する．対象とする制御の反転は com.jme.input.ActionTrigger クラスから Lesson8 中の Lesson8$2 クラスを呼び出すものとする．これは物体が衝突したときに発生する制御の反転である．

最終スタックにあらわれた参照から着目する参照を選び 4 節で述べた手法に基づいて再帰的にデルタ抽出を行った結果, 抽出されたデルタは 8 個, フレームワークの利用例は 3 例見つかった (表 1 参照). また, デルタ抽出の過程で確認された参照の総数は 20 だった. (詳細を表 2, 3 に示す.)

5.3 抽出結果の妥当性

表 1 に示したフレームワークの利用例が妥当なものであるのかについて確認を行った．まず eclipse については文献 12) に SWT サブシステムについての解説があり, その中に利用例に一致する記述があった．よって eclipse について得られた結果は妥当であると考えられる．jME については, 妥当であるか判断できるようなドキュメントが確認できなかったためサンプルコードの当該部分を 1 つずつ削除して実行することで動作がどう変わるか確認を行った．その結果, どの例においても今回対象とした制御の反転が発生しなくなることが確認された．よって jME についても得られた結果は妥当であると判断でき, 今回得られた全てのフレームワーク利用例は妥当であると考えられる．

6. 考 察

事例研究の結果を, 抽出結果の精度, 抽出作業の実効性の 2 つの観点から考察する．まず抽出結果の精度については, 偽陽性と偽陰性の 2 つの指標で評価する．偽陽性とは抽出した利用例に誤りが含まれていることを示し, 偽陰性とは抽出漏れがあることを示す．jME の事例では, サンプルアプリケーションが物理演算の機能のみを実装した単純なものであったため, そこから抽出した利用例は 5.3 節でも示したように偽陽性とならなかった．いっぽう

表 2 抽出された参照総数と着目参照数
Table 2 The total number and focused number of extracted references

		デルタ内参照			呼び出しスタック内参照		
		着目参照数	参照総数	削減率 (%)	着目参照数	参照総数	削減率 (%)
eclipse	最終スタック	–	–		3	11	
	Δ_1	0	3		0	15	
	Δ_2	0	0		0	0	
	Δ_3	2	2		1	5	
	Δ_4	0	1		0	0	
	Δ_5	1	1		0	0	
	Δ_6	0	3		0	0	
	Δ_7	0	0		0	0	
合計		3	10	70	4	31	87
jME	最終スタック	–	–		2	3	
	Δ_1	2	6		0	0	
	Δ_2	0	0		0	0	
	Δ_3	0	1		1	3	
	Δ_4	0	0		0	1	
	Δ_5	2	4		0	0	
	Δ_6	0	0		0	0	
	Δ_7	1	2		0	0	
	Δ_8	0	0		0	0	
合計		5	13	62	3	7	57

eclipse の事例では, 解析対象となったサンプルアプリケーションが jFace とそれを利用した eclipse のアプリケーション部分全体であったため, 抽出した利用例がアプリケーション固有の実装の影響を受ける可能性が高かった．本事例では, サンプルアプリケーション固有であることが明らかである CoolBar クラスや CoolItem クラスに関わる参照を追跡対象から除外したため, 結果的に 5.3 節で示したように偽陽性とならないようにコントロールすることができた．いっぽう偽陰性については厳密な検証は行っていないが, jME の事例では既知のプラグインポイントに関わる参照を追跡対象から除外し, そのプラグインポイントの抽出を省略したため, 得られた利用例は偽陰性となった．ただし本研究では, 偽陰性については大きな問題とはみなしていない．その理由は,

- 実際にアプリケーションを実装する文脈においては, 仮に抽出した利用例が偽陰性であった場合にその結果に基づいてアプリケーションを実装したとしても, 制御の反転が発生しない現象として容易に偽陰性が発生していることを確認することができる,
- さらに偽陰性を確認した場合, 追跡対象から除外した参照を再び追跡対象に加えて抽

© 2011 Information Processing Society of Japan

表 3 抽出された各デルタにおける参照の出現数
Table 3 The number of occurrences of references in eaxh extracted delta

		デルタ内参照	呼び出しスタック内参照
eclipse	最終スタック	–	11
	Δ_1	3	23
	Δ_2	0	24
	Δ_3	2	13
	Δ_4	1	12
	Δ_5	2	12
	Δ_6	4	22
	Δ_7	3	22
jME	最終スタック	–	3
	Δ_1	7	0
	Δ_2	0	5
	Δ_3	1	4
	Δ_4	0	2
	Δ_5	4	1
	Δ_6	0	1
	Δ_7	2	3
	Δ_8	0	1

出することによって, 不足しているプラグインポイントを後から追加抽出することができる,

の 2 点による.

次に抽出作業の実効性についてであるが, 表 2 で示したように新たに抽出される参照の多くを追跡対象から除外することによって, 抽出されるデルタの総数を現実的な範囲に抑えることができたため, 今回の事例研究では実行可能となった. ただし, 表 3 に示したデルタのうち概ねサイズ 4 以上のものをデバッガなどを用いて手作業で抽出するには多くの時間を要した. したがって, 抽出されるデルタの総数が増えたり, より大きなサイズのデルタの抽出が必要とされる事例では, 実効性に欠ける恐れがある. この問題については, 今後, デルタ抽出を自動化することで解決することができると考えられる.

7. 関連研究

フレームワークを正しく利用するためにはホットスポットの拡張に加えて, フレームワーク側のいくつかのメソッドを正しい場所で正しい順序で呼び出す必要があることが, いくつかの文献で指摘されている[5),6)]. 本研究が対象としているのはその中でもプラグインポイント[6)] のように, 一部でも欠落すればアプリケーション側で実装したメソッドが実行されなくなるといった強い制約を持つものである. これに対し, 文献 5) で指摘されているメソッド呼び出しは, 類似するコードの中で頻繁に出現するメソッド呼び出しのパターンを抽出して, アプリケーション開発者が記述したコードの逸脱度を示唆するためのものであり, 本研究や文献 6) が対象としているような強い制約を持つものではない. また, 文献 5) は統計的な手法に依拠しているため, 利用例を抽出するにはサンプルアプリケーション内の多くの個所で同一の利用例が出現していることが前提となるが, 本研究の手法ではサンプルアプリケーション内のたったひとつの出現箇所からでも利用例を抽出することができ, 加えてその利用例が必要である論理的な根拠も利用例と共に得ることができるという特長を持つ. さらに抽出精度については, 一般に統計的手法では抽出結果が偽陽性になる傾向が強いが, 本研究では抽出過程で得られるデルタの情報を用いて偽陽性にならないよう作業者が追跡対象をコントロールすることができる.

参照生成に関する依存関係を追跡する手法としてはオブジェクトフロー[9),10)] が良く知られている. オブジェクトフローを用いると, 同一のオブジェクトを参照している参照間の生成の依存関係を追跡することができる. しかしながら, オブジェクトフローでは依存関係はあるが異なるオブジェクトを参照している参照間の依存関係を追跡することができない. したがって, 仮にオブジェクトフローをフレームワークの利用例抽出に用いた場合, 制御の反転が依存しているにもかかわらず抽出できないようなメソッド呼び出しが生じる可能性がある. 実際に, jME の事例ではオブジェクトフローで追跡できない例が表 2 の Δ_5 の着目した参照の中に 1 つ存在した.

8. おわりに

プラグインポイントのように一部でも欠落するとアプリケーション側で実装したメソッドが実行されなくなるようなフレームワークの利用例を, サンプルアプリケーションの動的解析によって抽出する手法について提案を行い, その有効性と実効性を示すために 2 つのフレームワークについて事例研究を行った. 本研究で提案した手法では, 抽出した利用例が必要である論理的根拠を知ることができるため, アプリケーションの中でより安全に利用例を用いることができると考えられる.

今後の課題としてはデルタ抽出を自動化することによって, 全体の作業の効率化を図ることが考えられる. 特に大きなサイズのデルタの抽出を手作業で行うには多大な時間を要するため, 自動化による恩恵は非常に大きいと考えられる. また, 今回の事例では, 抽出された利

© 2011 Information Processing Society of Japan

用例をプラグインポイントに近づけるためにヒューリスティクスを用いて追跡対象のコントロールを行ったが, 今後コントロールの方法の体系化を行いたい.

参考文献

1) Mohamed E. Fayad, Ralph E. Johnson, Douglas C. Schmidt: Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons Inc (1999).
2) Douglas Kirk, Marc Roper and Murray Wood: Identifying and Addressing Problems in Object-Oriented Framework Reuse, *Empirical Software Engineering*, Vol. 12, No. 3, pp. 243–274 (2007).
3) Wolfgang Pree: Design Patterns for Object-Oriented Software Development. Addison Wesley (1995).
4) Marcel Brunch, Mira Mezini and Martin Monperrus: Mining Subclassing Directives to Improve Framework Reuse, *Proc. Working Conference on Mining Software Repositories*, pp. 141–150 (2010).
5) Martin Monperrus, Marcel Bruch, and Mira Mezini: Detecting Missing Method Calls in Object-Oriented Software, *Proc. of 24th European Conference on Object-Oriented Programming*, pp. 2–25 (2010).
6) George Fairbanks: Software Engineering Environment Support for Frameworks A Position Paper, *Proc. of WoDiSEE workshop at the International Conference on Software Engineering*, pp. 70–73 (2004).
7) Eclipse.org: eclipse, http://www.eclipse.org/, 販売者無し (2011).
8) Mark Powell ら: jMonkeyEngine, 販売者無し (2003).
9) Adrian Lienhard, Orla Greevy and Oscar Nierstrasz: Tracking Objects to Detect Feature Dependencies, *Proc. 15th International Conference on Program Comprehension*, pp. 59–68 (2007).
10) Adrian Lienhard, Tudor Gîrba and Oscar Nierstrasz: Practical Object-Oriented Back-in-Time Debugging, *Proc. 22nd European Conference on Object-Oriented Programming*, pp. 592–615 (2008).
11) 新田直也, 手塚裕輔: デルタ抽出: 実行履歴の大域的構造を効率良く把握するための抽象化手法, 情報処理学会研究報告, 2011-SE-173(6) (2011).
12) Erich Gamma and Kent Beck: Eclipse プラグイン開発, ソフトバンククリエイティブ (2004).

© 2011 Information Processing Society of Japan