Vol.2011-SE-173 No.6
2011/7/21

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

新田直也[†1]　手塚裕輔[†1]

プログラムの実行において, ある機能の実行が別の機能の実行に影響を及ぼす現象が多く見られる. そのため, 適切にデバッグや再利用を行うためには局所的なソースコードを読むだけでは不十分である場合が多く, 一般にソースコードに加えてプログラムの実行時の情報が参照される. しかしながら, 実用規模プログラムでは実行時に生成される情報が非常に膨大になり, その中から必要な情報を取り出すには多くの時間と労力を要する. そこで, 本研究ではオブジェクト指向プログラムの実行履歴から, プログラム理解に有用な大域的構造を効率良く取り出すための抽象化手法 (デルタ抽出と呼ぶ) を提案する. デルタ抽出の具体的な応用例としては, フレームワークの利用法抽出, 機能同定, バックインタイムデバッガなどを想定している. 本稿では, デルタ抽出に関する定義を与えるとともに, フレームワークの利用法抽出に対するデルタ抽出の適用可能性や有効性について議論する.

Delta Mining: An Execution Trace Abstraction for Efficient Comprehension of Its Global Structure

Naoya Nitta[†1] and Yusuke Tezuka[†1]

In an execution of a program, a pre-executed feature often affects a post-executed feature. Therefore, it is usually insufficient to only read a local part of the source code for appropriate reuse and debug tasks, and hence in addition to the source code, the runtime information of the program is also refered to. However, the amount of the runtime information tends to be quite large and mining useful information from it would require enormous time and effort. In this research, we present a method to efficiently mine a useful structure of an execution trace of an object-oriented program for appropriate program comprehension, and we call it delta mining. We suppose that delta mining can be applied for framework usage extraction, feature location and back in time debugger. In this paper, we define delta mining and discuss the applicability and the feasibility of the method for framework usage extraction.

1. はじめに

デバッグや再利用を適切に行う上で対象となるプログラムの理解は必要不可欠である. 一般にプログラムを理解する方法としては静的なアプローチと動的なアプローチがある. 静的なアプローチでは, ソースコードを直接読んだりツールを用いてソースコードの解析を行ったりすることによって, プログラムの理解が進められる. いっぽう, 動的なアプローチでは, コンパイルされたプログラムを実行し, 実行時に生成される情報を詳細に調べることによって, プログラムの理解が進められる. 実際の作業においては, ほとんどの場合両方のアプローチが組み合わされて用いられる. とりわけオブジェクト指向プログラムにおいては, 動的束縛などの処理機構によって呼び出されるメソッドが実行時に決定されるため, 動的なアプローチは必要不可欠である[8].

しかしながら, 一般に実行時に生成される情報は非常に膨大であり, その中から必要な情報を取り出すためには何度もプログラムの実行を余儀なくされる場合が少なくない. たとえば, デバッガを用いるとプログラムの実行を任意の時点で中断し, その時点での呼び出しスタックからアクセス可能なすべての変数の値を確認したり, ステップ実行によって制御の移動を 1 行ずつ追跡していくことも可能であるが, プログラムの過去の実行履歴は取得することができない. 文献 1) では, 不具合が発生した時点での呼び出しスタック中にその原因を特定できる情報が含まれていないケースが全体の約 5 割を占めることが報告されており, そのような場合, デバッガ上で不具合の発生を確認してもすでにその原因を特定するために有用な情報は失われており, 不具合の原因を特定するためには少なくとも 1 回以上のプログラムの再実行を余儀なくされる.

このような状況に対応するため, 本研究では Java プログラムを対象に, プログラムの実行のある時点で, あるオブジェクトがある変数によって参照されるようになった原因を過去に遡って辿ることができる動的解析手法としてデルタ抽出を提案する. 同様の手法としては, オブジェクトフロー[2)–4)] を挙げることができる. オブジェクトフローを用いると, 着目しているオブジェクトがどのような参照を経由して渡されてきたかという来歴を追跡することができる. ただし, オブジェクトフローで追跡することができるのはそのオブジェクトの流れ

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

に直接関与したコードのみであり, そのオブジェクトの流れを間接的に決定付けているコードについては追跡することができない. いっぽうデルタ抽出を用いると, オブジェクトの流れを間接的に決定付けているコードも含めオブジェクトの流れに寄与しているコードを網羅的に辿ることができる. これにより, デバッグだけでなく以下のようなアクティビティも支援することが可能になると考えられる.

- アプリケーションフレームワークのサンプルアプリケーションを解析して, フレームワークの利用法を抽出する. 具体的には, アプリケーション固有の機能を実装したオブジェクトがフレームワーク側の所定の位置で参照されるために, アプリケーション側に実装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網羅的に抽出する (フレームワークの利用法の抽出).
- 既存のプログラムのソースコードからある機能を実装している部分を抜き出す. 具体的には, ある機能固有のオブジェクトがその機能に依存しないオブジェクトに渡されるのに貢献したコードを網羅的に抽出する (機能同定).

本稿ではこれらの応用例のうちフレームワークの利用法抽出について具体的な事例を取り上げ, 本手法の適用を行った. その結果, 本手法を用いることによって, すべての作業を人手で行った場合と同等の結果をより体系的かつ効率的に得ることができることがわかった.

2. 問題事例

最初に, 実際のプログラム理解において直面する具体的な問題事例を紹介する. この問題事例は, ある目的のために著者の 1 人が十分な知識を持っていないアプリケーションフレームワークを約 1ヶ月の期間をかけて理解した過程を記録したものである. 本事例は, 5 節で説明する本手法の適用事例でも用いる.

問題の背景は以下の通りである. 本研究室では別の研究の中で, jMonkeyEngine[5] という 3D ゲームエンジンを用いて, あるゲームアプリケーションを実装できるか否かについて調査を行っていた. jMonkeyEngine は Java で書かれた 315KSLOC のオープンソースプログラムである. 当該アプリケーションでは “ゲームキャラクターが水平な地面の上を移動している最中に登り坂にさしかかった場合, その斜面の方向に移動を続ける” ことが要求されていた. このような振る舞いが jMonkeyEngine を用いて実装できるか否かを調べるには, jMonkeyEngine の物理演算機能の正しい利用方法を理解する必要がある. そこで, jMonkeyEngine の物理演算サブシステムに含まれていたサンプルプログラム Lesson8.java を詳細に解析することで, 物理演算機能の利用方法を調べることにした.

利用方法の実際の調査は, ソースコードの読解, ソースコードの改変とコンパイル, デバッガによる実行など静的アプローチと動的アプローチを適宜組み合わせて進められた. 作業全体として時間を要したのは, ブレークポイントを入れるべき場所の探索と, 実行時の詳細情報を確認しながらのデバッグ実行である. 最終的に理解するまでに, 合計で 60 箇所にブレークポイント, 7 箇所にウォッチポイントを入れた. その中で実際に利用方法を理解するために有効であったのは**表 1** に示す BP1∼BP9 のブレークポイント 9 箇所であった. また最終的に得られた物理演算機能の利用方法を**表 2** に示す. 利用方法は, 大きくフレームワーク側の型の継承とフレームワーク側のメソッドの呼び出しに大別することができる.

表 1 jMonkeyEngine の物理演算機能の利用方法の理解に必要なブレークポイント
Table 1 Breakpoints used to comprehend the usage of physical calculation in jMonkeyEngine

ブレークポイント	所属レイヤ	位置
BP1	アプリケーション	Lesson8$2.performAction()
BP2	基本フレームワーク	ActionTrigger.activate()
BP3	物理演算	ReusableContactInfo.set()
BP4	基本フレームワーク	ActionTrigger.ActionTrigger()
BP5	基本フレームワーク	UtilInputHandlerDevice.addButton()
BP6	基本フレームワーク	InputHandler.addDevice()
BP7	基本フレームワーク	UtilInputHandlerDevice.get()
BP8	物理演算	Geom.getGeomFromNativeAddr()
BP9	物理演算	Geom.retrieveNativeAddr()

表 2 得られた jMonkeyEngine の物理演算機能の利用方法
Table 2 Obtained usage of physical calculation in jMonkeyEngine

利用	対象レイヤ	内容
U1	基本フレームワーク	InputAction の具象クラスの実装
U2	基本フレームワーク	InputHandler.addAction() の呼び出し
U3	物理演算	PhysicsNode.getCollisionEventHandler() の呼び出し
U4	物理演算	DynamicPhysicsNodeImpl.generatePhysicsGeometry() の呼び出し

紙面の都合上, 理解の過程の詳細を説明することはできないが, 全体の概要は以下の通りである. まず最初に Lesson8.java のソースコードを見てアプリケーション固有のコードを実装すべき位置を確認した (U1). 次にこのコードが, 所望のタイミングでフレームワーク側から呼び出されるようにするために, アプリケーション側に実装しなければならないコード

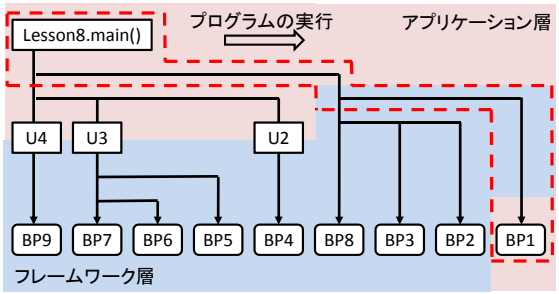

図 1 Lesson8.java の実行における呼び出し木
Fig. 1 A call tree of an execution of Lesson8.java

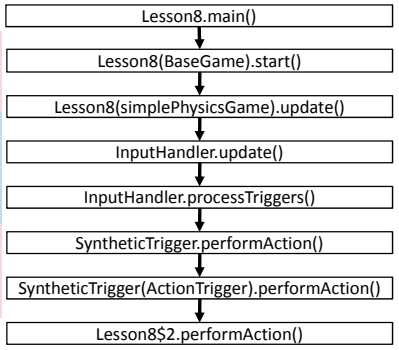

図 2 BP1 停止時の呼び出しスタック
Fig. 2 A call stack at BP1

を Lesson8.java を参考にしながら調べていくことにした．具体的には, Lesson8.java 内部に実装されているアプリケーション固有のコード (Lesson8$2.performAction()) にブレークポイント BP1 を入れてデバッガ上で Lesson8.java を動かし, プログラムが停止するまでに実行されたコードの中で, Lesson8$2.performAction() の呼び出しの実現に寄与したコードを実行とは逆向きに探索する．このとき呼び出しの実現に寄与したコードの探索は, 呼び出しの際に使われた参照の生成に直接的/間接的に関与したコードを探索することによって行われた．いずれにせよこれらの探索は, プログラムの実行とは逆向きに進める必要があるため, ブレークポイントを入れる場所を何度も変えながらプログラムを繰り返し実行する必要があった．Lesson8.java の起動から Lesson8$2.performAction() が呼び出されるまでに通過した主要なブレークポイントと, 各ブレークポイントに到達するまでの呼び出し木を**図 1** に, BP1 で停止した時点での呼び出しスタックを**図 2** に示す．U1 以外の利用方法は, この呼び出し木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抽出することで得られた．

3. デルタ抽出

3.1 デルタの概要

前節で指摘したように, プログラム理解の過程で, ある参照の生成に直接的/間接的に関与したコードの探索が必要になる場合がある．一般にあるオブジェクト o に対する参照の生成には, o を直接参照しているコードだけでなく, o を参照していないコードも関与しうる．たとえば, **図 3** に示したプログラムでは, 33 行の実行におけるクラス C のインスタンスへの新しい参照の生成には, 同じインスタンスを参照している 32, 19, 26, 24 行だけでなく, たとえばクラス B のインスタンスを参照している 12 行も関与している．なぜならば, 12 行で passB() メソッドに渡している B のインスタンスを別のものに変えると, 19 行で getC() メソッドによって取得される C のインスタンスも別のものに変わる可能性があるからである．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, 参照の生成に関与しているコード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参照が生成される．またその時のシーケンス図を**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

```
 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 サンプルプログラム
Fig. 3 An example program

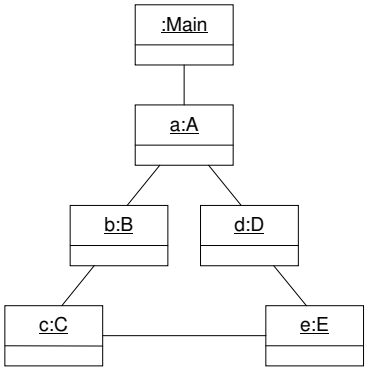

図 4 サンプルプログラムのオブジェクト図
Fig. 4 An object diagram of the sample program

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．直観的に参照 r のデルタは, r が生成されるために必要な事前条件を表す．

表 3 サンプルプログラムの実行における最小事前条件
Table 3 Minimum pre-conditions of an execution of the sample program

メソッド呼び出し	引数条件	フィールド条件	静的フィールド条件
E.setC(C c)	c	なし	なし
D.passB(B b)	b	this.e, b.c	なし
A.m()	なし	this.b, this.b.c, this.d, this.d.e	なし
Main.main()	なし	this.a.b, this.a.b.c, this.a.d, this.a.d.e	なし

3.2 デルタの定義

本節では, より厳密なデルタの定義を与える．3.1 節で説明したようにデルタは実行時に生成される参照 r に対して定義される．ここで, デルタの定義を与えるにあたって, まず r が生成されるための事前条件について考える．具体的には, 実行中に r が生成された時点から仮想的に実行の巻き戻しを行い, その時点でどのような条件が成り立っていれば, その後の実行で r が生成されるかについて考える．実行の巻き戻しは, r の生成時点で呼び出しスタック中に存在している各メソッド呼び出しの実行開始時点まで行う．たとえば, 図 3 の例では, E.setC(), D.passB(), A.m(). Main.main() のそれぞれの実行開始時点までの巻き戻しが考えられる．各メソッドの実行開始時点で r の生成に影響を与え得る条件として,

- そのメソッドの各引数の値,
- this オブジェクトの各フィールドの値,
- 引数およびフィールドの値がオブジェクトへの参照であった場合その参照から辿れる全オブジェクトの全フィールドの値,
- 全クラスの全静的フィールドおよび,
- 静的フィールドの値がオブジェクトへの参照であった場合その参照から辿れる全オブジェクトの全フィールドの値

が考えられる．これらの中で r の生成を左右するものを, 各メソッド実行開始時点における r の生成のための**最小事前条件**と呼ぶ．また最小事前条件のうち, 引数の値に関するものを**引数条件**, フィールドの値に関するものを**フィールド条件**, 静的フィールドの値に関するものを**静的フィールド条件**と呼ぶ．たとえば図 3 の例では, 各メソッド実行開始時点における最小事前条件は**表 3** のようになる．表 3 では, 実行の逆順にメソッド呼び出しを記載してい

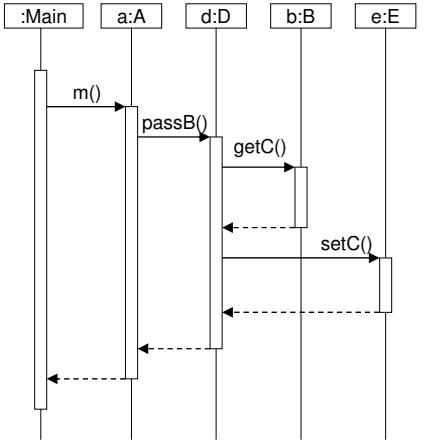

図 5 サンプルプログラムのシーケンス図
Fig. 5 A sequence diagram of the sample program

る．この表からわかるように, 基本的に過去に遡るにつれて引数条件がフィールド条件に置き換わっていき, ある時点 (この例の場合は, A.m() の呼び出し) まで遡るとフィールド条件は飽和する．そこで, r の由来を調べる上で, 引数条件とフィールド条件のいずれが好都合であるかについて考察する．まず, 引数条件はその引数の値の来歴を呼び出し元に向かって逐次追跡する必要があるため, r の由来を調べる上では不都合である．いっぽう, フィールド条件はその値の由来を調べる上でそのフィールドが最後に更新された時点まで実行を遡ることができるため好都合である．そこで, 実行の逆順にメソッド呼び出しを辿り, どの時点でフィールド条件が飽和するかについて考える．まず, 一般に引数条件のうちフィールド条件に置き換わるものと置き換わらないものがある．フィールド条件に置き換わるものは, その後の実行の中でその引数が示す参照を経由して r の参照元オブジェクトの ID もしくは r の参照先オブジェクトの ID を得ているという特徴を持つ．このような引数条件を**参照に依存する引数条件**と呼ぶ．参照に依存する引数条件に関して以下の命題が成り立つと考えられる．

命題 3.1 プログラムの実行中に生成される任意の参照 r について, r の生成時に呼び出しスタックに含まれているメソッド呼び出しの中で, 最小事前条件中に参照に依存する引数条件を含まないものが存在する．□

このとき, 上記命題を満たすメソッド呼び出しのうち最後に呼び出されたものを, r の**コーディネータ**と呼ぶ．また, コーディネータが実行されたオブジェクトを**コーディネータオブジェクト**と呼ぶ．コーディネータに関しては, さらに以下の命題が成り立つことが予想される．

命題 3.2 プログラムの実行中に生成される任意の参照 r について, r のコーディネータのフィールド条件および静的フィールド条件には以下のものが含まれる．

- コーディネータの実行開始時に r の参照元オブジェクト o_1 が存在するとき，コーディネータオブジェクトもしくはあるクラスオブジェクトから o_1 に向かうフィールドを介したパス p_1 の存在．
- コーディネータの実行開始時に r の参照先オブジェクト o_2 が存在するとき，コーディネータオブジェクトもしくはあるクラスオブジェクトから o_2 に向かうフィールドを介したパス p_2 の存在．□

ここで, 上記命題中の p_1, p_2, および r からなる構造を, r の**デルタ**と呼びその中に含まれている参照全体からなる集合を $\delta(r)$ で表す．また実行履歴から r のデルタを抽出することを r の**デルタ抽出**と呼ぶ．なお, この命題はコーディネータやそこから呼び出されたメソッドの実装内容に関わらず成り立つことに注意が必要である．これは, デルタが実行履歴の一部を抽象化していることを意味する．

例 3.1 3.1 節において説明した例を再掲する．たとえば, 図 3 のプログラムの実行において, オブジェクト e から オブジェクト c へのフィールド c を介した参照を r としたとき, r のコーディネータオブジェクトは オブジェクト a となり, r のコーディネータはそのインスタンスに対する m() の呼び出しになる．また r のデルタは, a.b, a.b.c, a.d, a.d.e および r によって構成される．□

4. デルタ抽出を用いた解析手法

実行履歴に対して繰り返しデルタ抽出を行うことで, 実行時に生成された任意の参照 r の生成条件を網羅的に探索することができる．まず r に対して抽出されたデルタを考える．このとき, 命題 3.2 より r が生成されるためには r のコーディネータの実行開始時点で $\delta(r)\backslash\{r\}$ に含まれる参照がすべて生成されている必要がある．さらに r のコーディネータが呼び出されるためには, そのコーディネータの実行開始時点の呼び出しスタックに含まれる全呼び出しで使用された参照がすべて生成されている必要がある．後者の条件をコーディネータへの**到達条件**といいその中に含まれている参照の集合を $\gamma(r)$ で表す．このとき, $\delta(r)\backslash\{r\}\cup\gamma(r)$ に属している任意の参照に対してデルタを抽出することができ, それらのデルタと r のデルタは接続しているという．形式的には**デルタ接続**は以下のように定義される．

定義 4.1 実行時に生成される任意の参照 r について, 別の参照 r' が $r' \in \delta(r)\backslash\{r\}\cup\gamma(r)$ を満たすとき, $\delta(r')$ と $\delta(r)$ は接続しているという．□

$r' \in \delta(r)\backslash\{r\}\cup\gamma(r)$ を満たす r' が生成されていることは, その後の実行で r が生成されるための必要条件であるため, $\delta(r)$ に接続するデルタを再帰的に抽出していくことで, 対象となる実行の中で r が生成されるための必要条件を網羅的に調べていくことができる．ただし, ここで求められる条件が r の生成のための十分条件でないことに注意が必要である．r のコーディネータの最小事前条件には, $\delta(r)$ に含まれないフィールド条件や, r に依存しない引数条件が含まれる可能性がある．そのため, 仮にデルタ抽出が自動化されたとしても, r が生成されるための条件を漏れなく探索するには, コーディネータ周辺のソースコードの読解は避けられない．ただし, そのような場合でも, 読むべきソースコードの範囲は大幅に局所化されると考えられる．コーディネータの周辺以外を読まなくて済む根拠となるのが前節で紹介した 2 つの命題である．まず, 命題 3.2 によって, 参照 r の生成条件を調べる上で r のコーディネータ実行開始から r の生成までに実行されたソースコードを読むことなく, 主に r のデルタと r のコーディネータへの到達可能条件のみに関心を集中させることができ

る．さらに，命題 3.1 によって，r のコーディネータの最小事前条件の中に少なくとも r に依存する引数条件が含まれないことが保障されるので，基本的にそれ以前の呼び出し履歴を辿る必要がなくなり，r のデルタに接続している別のデルタが出現する時点までソースコードを読むことなく実行履歴を遡ることができる (**図 6** 参照)．以上の理由により繰り返しデルタ抽出を行うことで，参照 r の生成条件の効率の良い探索が可能になることが期待されるが，デルタを再帰的に抽出していく過程で抽出されるデルタの数が膨らみ解析全体が実効的でなくなる可能性は残されている．そこで次節では，実際の解析の事例を通してデルタ抽出に基づく解析の実効性について検証を行う．

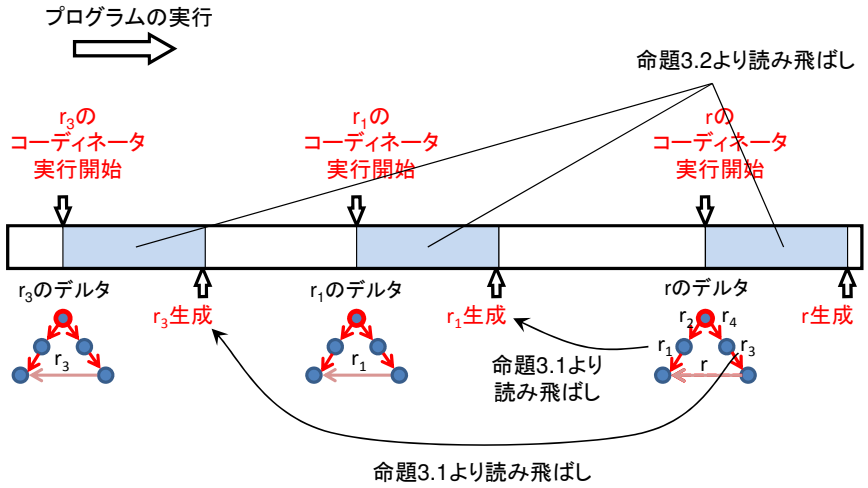

図 6 デルタ抽出を用いた解析手法
Fig. 6 Delta mining based analysis method

5. フレームワーク利用法抽出に対する適用事例

2 節で述べた問題事例について，デルタ抽出を用いて表 2 に示したフレームワークの利用法が抽出可能かどうか検証する．なお，この検証は 2 節で述べた作業を行った者が行った．デルタ抽出による解析過程を以下に示し，得られたデルタとその間の接続を**図 7** に図示する．図 7 ではオブジェクトを a〜o までの識別子で区別している．また，コーディネータオブジェクトを赤丸で示し，静的メソッドの呼び出しを破線矢印で示す．

(1) 実際の作業時と同様にブレークポイント BP1 を入れてデバッグ実行する．
(2) 実行が停止した時の呼び出しスタックの中で呼び出しに使われた 2 つの参照に着目する (それぞれ r_1, r_2 とする)．着目する参照の選別については解析者が判断した (以下も同様)．
(3) r_1 に対するデルタを抽出する (抽出されたデルタを Δ_1 とする)．
(4) Δ_1 から，U2 が得られる．
(5) さらに，Δ_1 中の 2 つの参照に着目する (それぞれ r_3, r_4 とする)．
(6) r_3 に対するデルタを抽出する (Δ_3 とする)．しかし，特に新しい情報は得られない．
(7) r_4 に対するデルタを抽出する (Δ_4 とする)．
(8) Δ_4 のコーディネータの呼び出しスタックから，U3 が得られる．
(9) Δ_4 に関して，新たに着目すべき参照は存在しない．
(10) r_2 に対するデルタを抽出する (Δ_2 とする)．
(11) Δ_2 のコーディネータの呼び出しスタック中で呼び出しに使用されたある参照に着目する (r_5 とする)．
(12) r_5 に対するデルタを抽出する (Δ_5 とする)．
(13) さらに，Δ_5 中の 2 つの参照に着目する (r_6, r_7 とする)．
(14) r_6 に対するデルタを抽出する (Δ_6 とする)．
(15) Δ_6 のコーディネータの呼び出しスタックから，U4 が得られる．
(16) 得られたデルタに関して，新たに着目すべき参照は存在しない．
(17) r_7 に対するデルタを抽出する (Δ_7 とする)．
(18) さらに，Δ_7 中のある参照に着目する (r_8 とする)．
(19) r_8 に対するデルタを抽出する (Δ_8 とする)．
(20) 得られたデルタに関して，新たに着目すべき参照は存在しない．

以上のようにして，デルタ抽出を用いた解析によって表 2 に示した U2〜U4 のすべてを得ることができた．また，解析者が着目する参照を選別することで抽出するデルタの数を 8 個に抑えることができた．

6. 考察と今後の課題

5 節の適用事例から，デルタ抽出を繰り返し行うことで，実際のフレームワークの利用法の抽出過程を再現できることがわかった．その際に，4 節で説明したように，デルタが持つ性

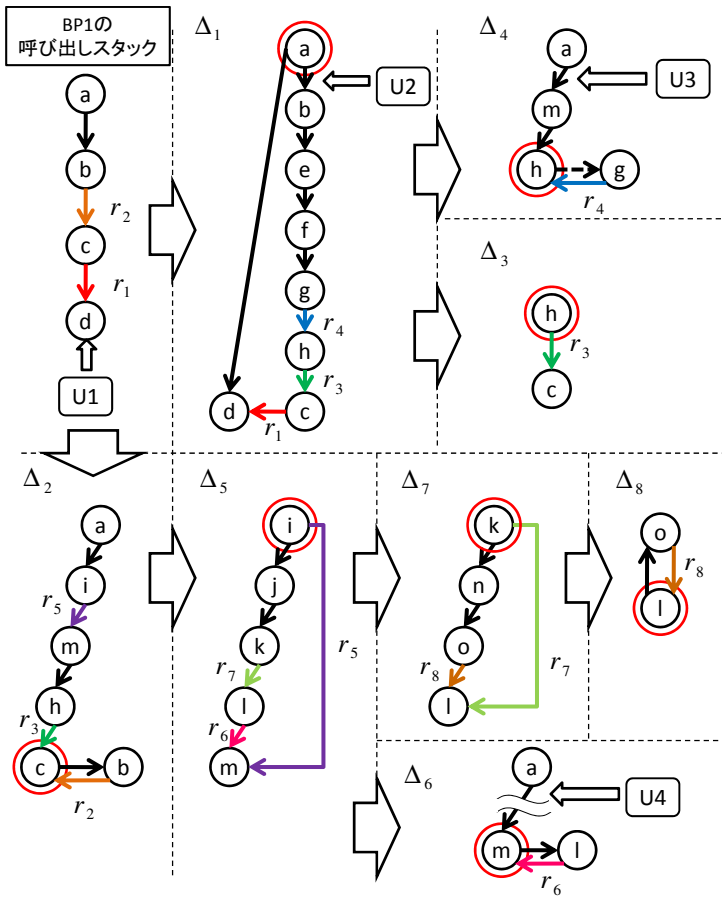

図 7 jMonkeyEngine のデルタ抽出を用いた解析例
Fig. 7 An analysis of jMonkeyEngine using delta mining

質を利用して読むべきソースコードの量を大幅に削減できることがわかった．さらに，同事例から実際にデルタを再帰的に抽出していく過程において，抽出されるデルタの数が際限なく膨らんでいかないこともわかった．これらのことから，デルタ抽出によるプログラムの理解支援は，フレームワークの利用法抽出においては十分に有効であると考えられる．今後他の事例にも適用することで，デルタ抽出の有効性について詳細に検証していく予定である．なお今回の適用事例では，現実のフレームワークの利用法抽出の際と同じように，解析者が必要に応じて関心のある参照の選別を行った．これによって，抽出されるデルタの数を大幅に抑えることはできたが，参照の選別にこのようなヒューリスティクスに基づかない戦略を定義できるかどうかは不明である．今後，解析者が実際に行った選別作業を詳細に調査することで，デルタの抽出戦略の構築を行いたい．なお今回の適用事例では，事前に得られた実際の解析結果を元に手動でデルタの抽出を行ったが，今後デルタ抽出のアルゴリズムを考案および実装して，デルタの抽出の自動化を目指す予定である．また，本稿で示した命題の厳密な証明を与えることも今後の課題である．なお，フレームワークの利用法抽出以外への応用として，ソフトウェア偵察法[9] などの手法と組み合わせることによって機能同定に応用することも検討している．

7. 関 連 研 究

あるオブジェクトの来歴を過去に遡って辿ることができる動的解析手法としては，オブジェクトフロー[2)–4)] が良く知られている．オブジェクトフローを用いると，着目しているオブジェクトがどのような参照を経由して現在の位置に渡されてきたかという来歴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対象とするオブジェクト o を固定して o への参照の伝搬を過去に遡るだけで，デルタが行っているような o への参照に影響を与えたが o 以外のオブジェクトを指しているような参照にまで追跡範囲を広げることができない．したがって，本研究が対象としているようなフレームワークの利用法抽出に適用した場合，抽出漏れが生じる可能性がある．

フレームワークの利用法抽出としては，メソッド呼び出し欠損の検出法[6)] を挙げることができる．この手法は，多数のフレームワークの利用例を解析して，フレームワークを利用する際に必要なメソッド呼び出しを統計的に検出するものである．しかしながら，本手法のようにたった一つの利用例から必要なメソッド呼び出しを検出することはできない．また，検出されたメソッド呼び出しが必要であることの根拠も得られない．デルタ抽出を用いた解析では，フレームワークを利用するための必要条件を実行履歴の中から洗い出していくため，抽出されたメソッド呼び出しの必要性は明らかである．

実行履歴を過去に遡ってデバッグできるようにするバックインタイムデバッガの研究が広く行われている[4),7)]．デルタを抽出するために必要な情報を残すことで，効率の良いバックインタイムデバッガを実現することができると考えられるが，具体的な実現方法については

今後の課題である.

8. おわりに

実行履歴を実行に対して逆向きに辿りながら進めるプログラム理解を系統的に支援する手法としてデルタ抽出を提案した. また, フレームワークの利用法抽出という問題に対して, デルタ抽出に基づく解析手法が有効であるか否かの事例研究を行った. その結果, 本手法を用いることで人手で得た結果と同一の結果を得られることがわかった. さらに, 読むべきソースコードの量を大幅に削減することで理解の過程を効率化できるという見通しを立てることもできた. 今後, より多くの事例に適用することで本手法の有効性を検証すると共に, 今回効率化のために人手に頼った判断を一貫した抽出戦略として定義していく予定である. また, デルタ抽出を行うアルゴリズムを考案し, 抽出の自動化を行うことも予定している.

参考文献

1) Liblit B., Naik M., Zheng A., Aiken A. and Jordan M.: Scalable Statistical Bug Isolation, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15–26 (2005).
2) Lienhard, A., Ducasse, S., Gîrba, T. and Nierstrasz, O.: Capturing How Objects Flow at Runtime, *Proc. International Workshop on Program Comprehension through Dynamic Analysis*, pp. 39–43 (2006).
3) Lienhard, A., Greevy, O. and Nierstrasz, O.: Tracking Objects to Detect Feature Dependencies, *Proc. 15th International Conference on Program Comprehension*, pp. 59–68 (2007).
4) Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugging, *Proc. 22nd European Conference on Object-Oriented Programming*, pp. 592–615 (2008).
5) Mark Powell ら, jMonkeyEngine, 販売者無し (2003).
6) Monperrus, M., Bruch, M. and Mezini, M.: Detecting Missing Method Calls in Object-Oriented Software, *Proc. 24th European Conference on Object-Oriented Programming*, pp. 2–25 (2010).
7) Pothier, G., Tanter, E. and Piquer, J.: Scalable Omniscient Debugging, *Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 525–552 (2007).
8) Wilde, N. and Huitt, R.: Maintenance Support for Object Oriented Programs, *IEEE Trans. Softw. Eng.*, Vol. 18, pp. 1038–1044 (1992).
9) Wilde, N. and Scully, M.: Software Reconnaissance: Mapping Program Features to Code, *Journal of Software Maintenance: Research and Practice*, Vol. 7, pp, 49–62, Jan (1995).