Javaプログラムを対象とした動的解析リポジトリの自動生成

大須賀　慎一[†]　　大久保　弘崇[††]　　粕谷　英人[††]　　山本　晋一郎[††]

[†]愛知県立大学大学院 情報科学研究科　　[††]愛知県立大学 情報科学部

Automatic Generation of Dynamic Analysis Repository for Java

Shinichi OHSUGA[†]　Hirotaka OHKUBO[††]　Hideto KASUYA[††]
Shinichiro YAMAMOTO[††]

[†]Graduate School of Information Science and Technology, Aichi Prefectural University

[††]Faculty of Information Science and Technology, Aichi Prefectural University

1　はじめに

ソフトウェアの再利用や保守において、ソフトウェアの動作理解は必要不可欠である。しかし、ソフトウェアには多くのソースプログラムが含まれており、その全貌を理解し、振る舞いを捉えることは容易ではない。このため、開発者や保守作業者の作業量を軽減する理解支援ツールが注目されている。

現在利用されているプログラム理解支援ツールの多くはソースプログラムを静的に解析した情報を扱っているが、静的な手法による解析情報だけではプログラム理解には不十分である場合も多い。

理解支援のためのより多くの情報を得るには、プログラムを実際に実行して解析を行う動的解析を共に利用することが有効であると考えられる。

本稿では、Java プログラムの動的解析を対象に、再利用可能な動的情報リポジトリを提案する。デバッガなどの一般的な動的解析ツールでは実際使用しているメモリの値を参照するため、情報の揮発性が高く後戻り不可能である。そのため、値の変動に対して人間の理解が追いつかなかった時などは、はじめから実行しなおす必要がある。そこで、一回の実行でコールグラフや変数履歴などの動的情報を取得・保存して、後から利用可能なリポジトリを生成する。このリポジトリを提供することでプログラム理解のための CACE ツール作成を支援する。リポジトリ形式は容易に利用できるようにするため、多くのパーサーが提供されている XML[1] 形式とした。

本稿の構成は、2 章でプログラム理解と解析についてを述べる。3 章では、プログラムの動的情報モデルを定義する。4 章では動的情報モデルを使用したプログラムの動的解析システムを実装し、5 章で評価する。最後に 6 章でまとめと今後の課題について述べる。

2　プログラム理解と解析

2.1　プログラム理解

ソースプログラムは、計算機上で実行するために記述されたものであるので、完成したプログラムであっても、人間の理解に適した形になっていない。たとえば、メソッド名やコメントが適切に与えられていないソースプログラムを人間が理解するのは容易ではない。

しかし、プログラム保守のための修正作業や、類似したプログラムを作成する際に、再利用したり参考にするため、ソースプログラムを理解したいという要求がある。

そこで、ソースプログラムを解析し、人間が理解しやすい形のビューを与えることで、プログラム理解を支援する手法が研究されている。

2.2　プログラム解析

プログラムの解析は大きく分けて静的解析と動的解析の二つに分類される。以下でそれらについて簡単に説明する。

2.2.1　静的解析

静的解析は、プログラムを実行せずに、ソースプログラムのみを解析する手法であり、実行時の入力に依存しない部分についての解析を行う。

静的解析には、関数呼び出し関係 (static call graph) や、変数の定義・参照関係の解析などがある。

2.2.2　動的解析

動的解析とは、プログラムに与えられる入力などの外部環境についても考慮し解析を行うことで、一般的には実際にプログラムを実行し、変数値や実行パスについての情報を取り出し解析する。

代表的な動的解析ツールであるデバッガでは、プログラムソースの途中にブレークポイントと呼ばれる印をつけてプログラムを実行すると、その地点でプログラムの動作を一時停止し、変数値やコールスタックの値を確認することができる。C 言語を対象とした代表的なものに GDB[2] があり、Java 言語にも jdb というデバッガが JDK[3] に用意されている。

2.3　Javaプログラムにおける動的解析

動的解析についての詳細を Java プログラムを対象に以下で述べる。

2.3.1 コールグラフ (dynamic call graph)

コールグラフとは、実際にプログラムを動作させたときのメソッド呼び出し順序や入れ子関係の情報を持つグラフである。したがって、解析時に呼び出されなかったメソッドはグラフに現れない。

一方、静的解析における関数呼び出し関係 (static call graph) はソースプログラムに記述されている呼び出し関係をすべて抜き出したもので、順序や入れ子関係は完全には解析することができない。

このコールグラフを利用することによって、初期化や後始末が適切に行われているかどうかの検査を行うことができる。

2.3.2 変数値履歴

プログラム中の変数は時間とともに変化する。その変化のタイミングと新たな値の履歴を保存することによって、プログラム動作中の任意の時間においての変数値を取得することができる。

2.3.3 メソッド引数・返り値

メソッド呼び出し時の引数と返り値はメソッドにおける入出力であり、それらが仕様に合っているかどうかの検査を行うことができる。

2.3.4 オブジェクト参照関係

Java はオブジェクト指向言語なので、オブジェクトに関連した動的情報も存在する。オブジェクト参照関係は、オブジェクトがいつどこで生成され、どこから参照されているかの情報である。生成を知るためには new 演算、参照を追うためにはオブジェクトの代入を監視すれば情報を得ることができる。

2.3.5 例外

例外が発生すると、例外の種類を示すオブジェクトが生成され、直ちに例外処理部分に実行が移る。そのため、コールグラフとオブジェクト参照関係に密接な関係がある。例外が発生した個所、投げられたオブジェクト・例外処理が行われた個所を把握する必要がある。

3 Java プログラムの動的モデル

図 1 に本稿で提案する動的解析モデル、**DADB**(Dynamic Analysis DataBase) のクラス図を示す。なお、動的情報は時系列順に格納されていくので、すべての集約関係はオーダードコレクションである。以下で各クラスの詳細について説明する。

DADB クラスは動的情報本体の情報を持つ Action クラスとオブジェクトやクラスの ID 表を持つ、IDTable クラスの 2 つを持つ。

Action クラスはメソッド呼び出しや代入といったプログラムの動作についての情報の全体を表し、Event クラスの集合からなる。

IDTable クラスは、Java プログラムのクラス・メソッド・オブジェクトについての情報を保持するクラスで、Action クラス以下から IDREF で参照される。その詳細は 3.2 節で述べる。

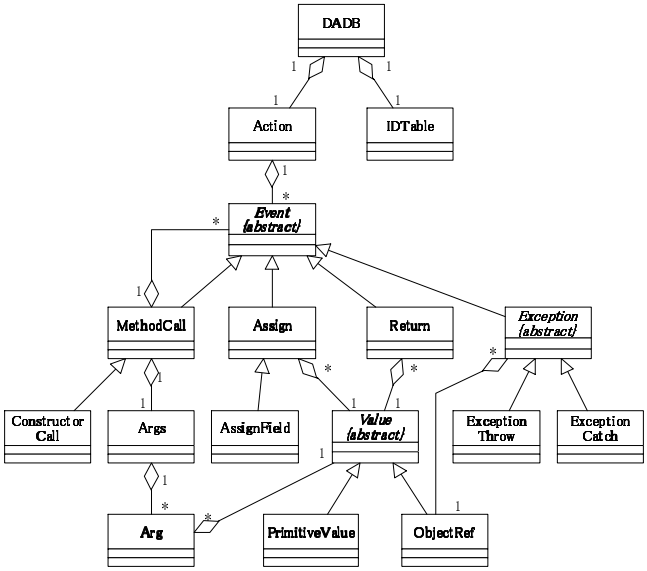

図 1: DADB モデルのクラス図

3.1 プログラム動作情報クラス

Action クラスの内部で利用されるクラスで、性質別に 3 つに分割した。これらのクラス群が実質的な動的情報を表している。

3.1.1 動的イベント関連クラス

プログラムを動的に捉えるとプログラム動作の最小単位として、メソッド呼び出し、代入、オブジェクト生成、return、exception が存在する。本稿ではそれらを動的なイベントと呼んでいる。

以下で動的なイベントを示すクラスを説明する。

- Event クラス
  動的なイベントのスーパークラス。全てのイベントはプログラムソース上どの行で発生したかの情報を持ち、発生順に通し番号が振られている。これによりいつどこでそのイベントが発生したかを調べることができる。
- MethodCall クラス
  メソッド呼び出しを表し、引数についの情報を持つ Args クラスと Event クラスを持つ。メソッド呼び出し時の引数の値の他、リポジトリから全ての MethodCall および ConstructorCall クラスを取り出すことによって、コールグラフを得ることができる。また、局所変数についてのスコープの役目も担っている。
- ConstructorCall クラス
  コンストラクタ呼び出しを表し、new によるオブジェクト生成を同時に意味する特殊な MethodCall。上記のコールグラフ取得の他に、オブジェクトがいつ生成されたか調べることに利用できる。
- Assign クラス
  局所変数に対する代入を表し、代入された値を示す Value クラスを 1 つ持つ。1 つの MethodCall クラスが持つの

Assign クラスを取り出すことにより、そのメソッド内の局所変数の値履歴を取り出すことができる。

- AssignField クラス
  フィールドへの代入は AssignField クラスとして扱う。代入された値を示す Value クラスを持つ。フィールドはオブジェクトごとに存在するので、所属するオブジェクトを示す ID が属性として与えられている。特定のオブジェクト ID を持つ AssignField を調べることにより、そのオブジェクトの各フィールドについて値履歴を取得することが可能である。
- Return クラス
  返り値を表し、その返却値を示す Value クラスを持っている。MethodCall クラスの持つ Args クラスと Return クラスを調べることにより、メソッドへの入力とその出力を得ることができる。
- Exception クラス
  例外に関する情報を意味する ExceptionThrow と ExceptionCatch のスーパークラスである。Java の例外はオブジェクトを投げ渡しすることが可能なので、Exception クラスは ObjectRef クラスを持つ。
- ExceptionThrow クラス
  例外が発生したことを示し、その際に投げるオブジェクトを保持している。
- ExceptionCatch クラス
  例外の受け取りを示し、ExceptionThrow と同様に受け取ったオブジェクトを持つ。

3.1.2 引数関連クラス

- Args クラス
  0 個以上の Arg クラスを持ち、メソッド呼び出し時の引数の集合を表す。
- Arg クラス
  メソッド呼び出しにおける個々の引数を表し、渡された値を Value クラスとして持つ。これにより、メソッド呼び出し時の引数の値やオブジェクトを調べることができる。

3.1.3 値関連クラス

- Value クラス
  プリミティブ型の値とオブジェクト参照のスーパークラスで、値を表す。
- PrimitiveValue クラス
  プリミティブ型の値を表す。
- ObjectRef クラス
  オブジェクトへの参照を表す。Java プログラムでは本来参照という言葉は用いないが、オブジェクトの扱いは C++ プログラムでいうところの参照やポインタに近い。そのため、本研究では、オブジェクトは参照であるとして抽象化する。

3.2 IDTable 構成クラス

IDTable クラスはクラス名などの静的な情報と ID の対応表である。

- Class クラス
  'c' から始まる数字の ID が振られ、クラス名と定義されているファイル名を属性として持つ。
- Method クラス
  'm' から始まる数字の ID が振られ、メソッド名を属性として持つ。
- Object クラス
  'o' から始まる数字の ID が振られている。

各 ID は JVMTI がプログラム動作中に振った値をそのまま用いている。この表を参照することによって、メソッド名やどのオブジェクトがどのクラスとして生成されたか、どのファイルで定義されたクラスかなどの情報を得ることができる。

これらのクラスは Action クラス以下のクラスから参照されている。Action クラスと IDTable クラスの関係を図 2 に示す。破線で囲まれているのが図 1 から関連するクラスのみを抜き出した、Action クラス側のクラスである。

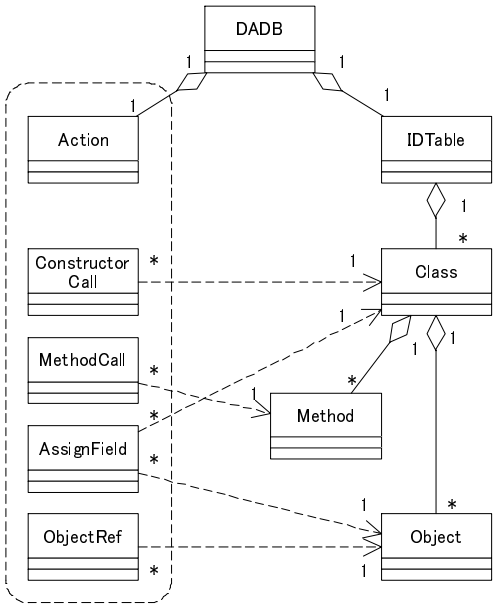

図 2: DADB のクラス図 (IDTable 部分)

4 実装

前章で提案した DADB モデルに従い、JVMTI[4] を用いて Java 仮想マシンから情報を取得し、時系列順に XML 形式のリポジトリに格納するプログラムを作成した。

リポジトリ生成の流れを図 3 に示す。

4.1 JVMTI による情報の自動取得

JVMTI(Java Virtual Machine Tool Interface) とは、JDK に含まれるツールの 1 つであり、従来の Java Virtual Machine Profiling Interface と Java Virtual Machine Debug Interface

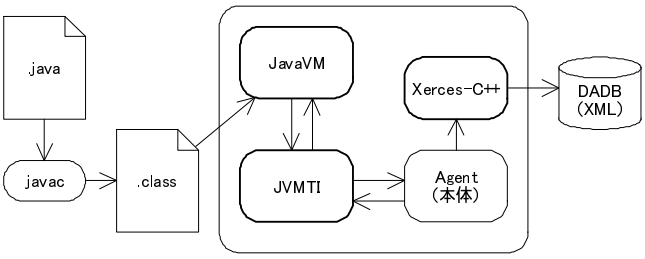

図 3: リポジトリ生成の概略

を統合した、開発ツールおよび監視ツール用のネイティブプログラミングインタフェースである。Java 仮想マシン (VM) で動作するプログラムの状態検査と実行制御の両方の機能を提供している。

JVMTI のクライアントは “Agent” と呼ばれ、VM 内で発生するメソッド呼び出しなどの様々なイベントを受け取り解析することが可能である。Agent を用いることで、既存のソースプログラムには手を加えずに動的な解析を行い、情報を取り出すことができる。Agent の作成には、C/C++言語をサポートする任意のネイティブ言語が使用できる。

このインタフェースを用いて、前章で提案した DADB モデルに従って動的情報を取得する Agent を C++を用いて実装した。

4.2 XML への格納

DADB モデルのクラス関係を XML で表現するため、DTD(文書型定義) を作成した。DADB モデルの各クラスを XML ノードとして扱い、ノードの親子関係を集約関係、実体宣言を汎化関係として利用している。

Agent で得られた動的な情報を C++言語用の XML パーサの一つである Xerces[5] を用いて DTD に沿って XML に格納していき、リポジトリを生成する。

5 リポジトリの評価

本稿で提案したリポジトリの評価のために、実際にリポジトリを利用した CASE ツール ELC(Executed-line Counter) を作成した。

ELC はプログラムソースの行ごとの実行回数を動的情報リポジトリから XML パーサを用いて解析し、その情報をマークアップした XML 形式ソースプログラムを生成するツールである。

このマークアップしたソースプログラムは XSL を用いてブラウザで表示すると、実行された行が実行頻度別に色付けされたソースプログラムとして閲覧できる。そのため、プログラムの分岐やループの様子を視覚的に捉えることができる。図 4 にそのサンプルを示す。

動的情報の取得はリポジトリから取り出すだけなので、このツールは XML パーサのライブラリを除くと、300 行程度プログラムで実装することができた。

6 むすびと今後の展望

本稿では、Java プログラムの動的な解析情報を保存・活用するためのリポジトリを提案した。そのために、Java プログラム

```
1:
2: class Test3 {
3:     public static void main(String argv[]) {
4:             Owner obj = new Owner();
5:
6:             if (argv.length > 1) {
7:               obj.setAct(new NormalAction());
8:             }
9:             obj.start();
10:    }
11: }
12:
13: class Owner {
14:    Owner() {
15:            act = new DefaultAction();
16:    }
17:    void setAct(Action a) {
18:            act = a;
19:            init();
20:    }
21:    void init() {
22:            act.init();
23:            status = 0;
```

図 4: XSL 変換による出力

における理解支援に必要な動的情報をまとめ、プログラム実行時に JVMTI を用いて自動的に動的情報を取得し、リポジトリを生成する Agent を作成した。

また、実際にこのリポジトリを利用したツールの作成を行った。これにより、生成した動的情報リポジトリを後から利用することによって、VM からの煩雑な動的情報の取得を必要とせず、XML パーサの利用だけで手軽に動的解析を行えることを示した。

今後の課題は、静的解析情報との統合である。今回の解析では動的情報を利用しやすくするために、Java コンパイラのデバッグオプションによって中間コードに埋められる、クラス名や変数名といった最低限の静的情報しか含んでいない。しかし、静的解析を行う外部のツールと組み合わせることによって静的解析と動的解析を併用した、よりプログラム理解に役立つツールを作成することが可能であると考えられる。

参考文献

[1] XML : Extensible Markup Language
http://www.w3.org/XML/

[2] GDB : The GNU Project Debugger
http://sources.redhat.com/gdb/

[3] JDK : Java Developers Kit
http://java.sun.com/j2se/1.5.0/ja/docs/ja/

[4] JVMTI : Java Virtual Machine Tool Interface
http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jvmti/

[5] Xerces C++
http://xml.apache.org/xerces-c/

[6] 日高 隆博, “プログラム動作情報のモデル化と理解支援への応用”, 名古屋大学大学院工学研究科 情報工学専攻, 1999

[7] 鈴木 孝聡, 山本 晋一郎, 阿草 清滋, “Java プログラムの振舞いのモデル化”, 日本ソフトウェア科学会 FOSE’99, pp.236–243 (1999/11)