SOBA: Javaバイトコード解析ツールキット

石尾 隆[1,a)]　秦野 智臣[1,b)]　井上 克郎[1,c)]

概要：Java を対象としたプログラム解析の研究では，制御フローグラフやそれに基づく依存関係，メソッドの呼出し関係がしばしば必要になる．本稿で紹介する SOBA (Simple Objects for Bytecode Analysis) は，それらの情報に迅速にアクセスできるようにするクラスライブラリである．

1. はじめに

プログラム解析は，開発されたソフトウェア製品から有益な情報を抽出し，開発者に提供するための重要な基盤技術である．たとえば製品のソースコードから得られる行数や複雑度といった特徴量は品質管理の基本情報として知られている．また，プログラム中での呼び出し関係やクラスの継承関係といった情報は，プログラムの構造を理解するために役立つ情報として，統合開発環境などを通じて開発者に提供されている．

プログラム解析を実現する基本的な方法の 1 つは，コンパイラと同様にソースコードを対象とした構文解析を実行し，抽出した構文木や記号表の情報を用いるというものである．既存研究で開発されたツールとしては MASU [1] や UNICOEN [2] があり，いずれも複数のプログラミング言語の解析に対応するための工夫が凝らされている．

プログラム解析のもう 1 つの方法が，コンパイルされたバイナリの解析である．Java のバイナリ表現であるバイトコードの場合，ソースコードの行数のような情報は正確には得られないが，参照するクラスのパッケージ名やメソッドの引数の型情報など，ソースコードには直接出現しないコンパイラによって追加された情報を解析に利用できる．このアプローチは Soot [3] などに採用されている．

SOBA は，我々の研究グループで開発した，Java のバイトコードに対する基本的な解析機能を提供するツールキットである．メソッド内部の命令の実行順序を取得する制御フロー解析，その情報を用いた制御依存関係解析とデータ依存関係解析，クラス階層を使用して動的束縛を解決する Class Hierarchy Analysis (CHA) [4] と Variable Type Analysis (VTA) [5] の実装を提供している．実装としては ASM [*1] のラッパーであり，論文執筆時点で JDK 1.8 までのバイトコードの解析が可能である．SOBA 自体のライセンスとしては MIT ライセンスを採用し，ソースコードを OSDN にて公開している [*2]．Soot と比べると実装された解析アルゴリズムの数は少ないが，詳細な事前知識なしで利用できるものを提供している．

2. SOBA の設計方針とプログラム例

SOBA は，コンパイルされた Java プログラムの解析を Java で記述するためのツールキットである．プログラムに含まれるクラスやメソッドの一覧，メソッドの呼び出し関係やクラス階層といった基本的な情報に，プログラム解析を初めて行う人でも手軽にアクセスできるようにすることを目的としている．

SOBA では，利用者の習得を容易にするという観点から，プログラムを「読み込む」指示を行うだけで，以下のような階層的なデータ構造を構築するようになっている．

- プログラム全体を表現する `JavaProgram` オブジェクト
- プログラム中の各クラスを表現する `ClassInfo` オブジェクト
- クラス中の各メソッドを表現する `MethodInfo` オブジェクト

利用者はプログラム例などを参考にして `JavaProgram` オブジェクトの作成さえできれば，そこから `getClasses()`，`getMethods()` といった `get` メソッドを呼び出していくだけで，クラス，メソッドの基本情報や，メソッドの制御フローグラフなどの解析に必要なオブジェクトを取得することができる．

SOBA を使用したプログラムの一例として，Java プログラムからメソッド呼出し関係を抽出する方法を**図 1** に示

[1] 大阪大学
Osaka University, Suita, Osaka 565-0871, Japan
[a)] ishio@ist.osaka-u.ac.jp
[b)] t-hatano@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

[*1] `http://asm.ow2.org/`
[*2] `http://osdn.jp/projects/soba/`

```
public static void main(String[] args) {
  // (1) 引数で指定されたディレクトリ/zip の内容を読み込む
  JavaProgram program = new JavaProgram(
    ClasspathUtil.getClassList(args));

  // (2) クラス階層情報を取得する
  ClassHierarchy ch = program.getClassHierarchy();

  // (3) 全クラスの全メソッドの呼び出し命令を列挙する
  for (ClassInfo c: program.getClasses()) {
    for (MethodInfo m: c.getMethods()) {
      System.out.println(m.toLongString());
      for (CallSite cs: m.getCallSites()) {
        // (4) 動的束縛を解決し，結果を出力する
        MethodInfo[] callees = ch.resolveCall(cs);
        if (callees.length > 0) {
          for (MethodInfo callee: callees) {
            System.out.println("  [inside] " +
                               callee.toLongString());
          }
        } else {
          System.out.println("  [outside] " +
                             cs.toString());
} } } } }
```

図 1 プログラム内部でメソッド呼び出し関係を列挙する例

す．このプログラムの先頭部分 (1) では，解析したい Java のバイトコードが格納されたディレクトリや jar ファイルがコマンドライン引数として指定されると想定して，その内容をもとに JavaProgram オブジェクトを作成している．

動的束縛の解決には Java のクラス階層情報が必要であるが，これも JavaProgram オブジェクトから取得できる．図 1 における (2) で取得している ClassHierarchy クラスが，クラス階層と Java 言語仕様に従って呼び出しうるメソッドを列挙する CHA [4] の実装である．

Java プログラムからその内部のメソッド呼び出し関係を取得する場合，全クラスの全メソッドについて，すべてのメソッド呼び出しを列挙し，その呼び出し先を（動的束縛を解決して）出力すればよい．プログラムを表現した階層的なオブジェクト構造を先に述べたような手順で訪問する処理が図 1 における (3) のループ構造である．

メソッドを表現する MethodInfo クラスには，内部で呼び出すメソッドの一覧や，制御フローグラフや制御依存関係，データ依存関係などを取得する様々な get メソッドが用意されている．getCallSites はメソッド呼び出し命令の位置とその内容を取り出すメソッドであり，(4) の部分で動的束縛の解決を行って，実際の呼び出し先の情報を取得し，文字列として書き出している．これは，リフレクションの利用による呼び出し関係を含まない，一般的なコールグラフの情報に対応する．

動的束縛の解決には，ライブラリを含むすべてのクラスの情報が必要である．ライブラリもバイナリの一種であるため，解析時に引数としてライブラリのファイルも合わせて指定するだけで解析を実行することができる．ライブラリ内部の呼び出し関係は出力しないといった特別扱いをしたい場合は，(1) の段階でライブラリであることを明示的に指定し，ClassInfo，MethodInfo クラスの isLibrary メソッドを使用してコードを記述することになる．

動的束縛の解決に VTA [5] を使用する場合は ClassHierarchy クラスを CallResolver クラスに入れ替えるだけでよい．このコード例と VTA の利用例は，いずれも SOBA のリポジトリに収録されている．

なお，バイトコード解析は，ソースコード解析に比べて高速な解析を実現しやすい．たとえば，Eclipse IDE for Java Developers 4.4.1 と Oracle JDK 1.8.0 の合計 99,770 クラス，774,261 メソッドからすべてのメソッド呼び出し関係を抽出すると，その処理時間は Intel Xeon E5-2620 2.0GHz の 1 スレッドで約 42 分，1 メソッドあたりで 3.2 ミリ秒となる（ただし結果を文字列として書き出す時間を除く）．

3. 開発の状況

SOBA は，我々の既存研究（たとえば [6]）で使用しており，多くの解析の基盤となるような基本機能を備えていると考えているが，習得容易な API を目指すという点でまだ発展途上にある．プログラム解析を必要とする研究者，技術者に利用していただき，フィードバックがいただけると幸いである．

謝辞　本研究は JSPS 科研費 No.26280021 の助成を受けたものです．

参考文献

[1] 三宅達也，肥後芳樹，楠本真二，井上克郎：多言語対応メトリクス計測プラグイン開発基盤 MASU の開発，電子情報通信学会論文誌 D，Vol. J92-D, No. 9, pp. 1518–1531 (2009).

[2] 坂本一憲，大橋　昭，太田大地，鷲崎弘宜，深澤良彰：UNICOEN: 複数プログラミング言語対応のソースコード処理フレームワーク，情報処理学会論文誌，Vol. 54, No. 2, pp. 1234–1249 (2013).

[3] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot - a Java Bytecode Optimization Framework, *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, p. 13 (1999).

[4] Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis, *Proceedings of the 9th European Conference on Object-Oriented Programming*, pp. 77–101 (1995).

[5] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E. and Godin, C.: Practical Virtual Method Call Resolution for Java, *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 264–280 (2000).

[6] Hatano, T., Ishio, T., Okada, J., Sakata, Y. and Inoue, K.: Extraction of Conditional Statements for Understanding Business Rules, *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice*, pp. 25–30 (2014).