SOBA: シンプルな Java バイトコード解析ツールキット

秦野 智臣　石尾 隆　井上 克郎

ソフトウェア工学において，Java を対象としたプログラム解析を行う場合，制御フローグラフやデータ依存関係，制御依存関係，メソッドの呼び出し関係がしばしば必要になる．これらの情報を得る手段として，既存のプログラム解析ツールを用いることが考えられるが，利用者が解析アルゴリズムについて十分理解していないと，その利用が困難である．SOBA(Simple Objects for Bytecode Analysis) は，プログラム解析に関する詳細な知識なしで容易に利用できるように設計された Java バイトコード解析ライブラリである．本論文では，SOBA の主要な機能や設計方針，プログラム例を紹介する．

For program analysis of Java in software engineering, control-flow, data dependence, control dependence, and method call relationships are often required. Existing tools provide features to extract the required information, however, it is difficult to use the tools for users who are unfamiliar with algorithms for program analysis. SOBA (Simple Objects for Bytecode Analysis) is a class library that is easy to use without detailed knowledge of program analysis. This paper introduces main features, design policy, and example programs of SOBA.

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 [9] や UNICOEN [11] 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 [13] などに採用されている．

既存ツールを用いてプログラム解析を行うためには，利用者は解析アルゴリズムの動作や性質を理解していなければならない．たとえば，Soot には豊富なオプションが存在するため，利用者は各オプションを選択するための知識が要求される．ソフトウェア工学において，アルゴリズムに関する詳細な知識を持たない人がプログラムの情報を取得したい場合には，既存ツールを利用することが困難である．

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

SOBA: A Simple Toolkit for Java Bytecode Analysis.
Tomomi Hatano, Takashi Ishio, Katsuro Inoue, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.
コンピュータソフトウェア, Vol.33, No.4 (2016), pp.4–15.
[ソフトウェア論文] 2015 年 10 月 30 日受付.

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，動的束縛の解決などの機能を提供しており，これらの機能を詳細な事前知識なしで利用できるように設計されている．実装としては，Java バイトコードの解析ライブラリである ASM †1 のラッパーとなっており，2015 年 10 月の時点で 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．OSDN では，本論文に掲載しているコード例も公開している．SOBA はプログラム解析の基盤として活用することができ，我々の研究グループで利用している [4] [5] [6] [8].

以降，2 章では開発の動機に当たる既存のプログラム解析ツールの特徴について述べ，3 章では SOBA の実装について説明する．4 章では SOBA と既存ツールの Soot，WALA との比較を行う．5 章ではまとめと今後の課題について述べる．

2 開発の動機

2.1 Java バイトコード解析ツール

ソフトウェア工学の様々な研究で活用されている Java バイトコード解析ツールとしては，Soot [13] と WALA [14] の 2 つが挙げられる．これらのツールには，プログラム解析に利用できる様々なアルゴリズムが実装されており，たとえばコールグラフや制御フローグラフなどを取得することができる．

これらの既存ツールは，プログラム解析の様々なアルゴリズムを備えている一方で，利用者が解析アルゴリズムについて理解していないとツールを実行すること自体が困難であったり，実行された内容を理解することが困難であったりする．たとえば，Soot を利用してコールグラフを取得する場合は，ポインタ解析アルゴリズムを選択したり，ポインタ解析を行うための二分決定木の実装を選択したりする必要がある．アルゴリズムによって解析結果の正確さや実行時性能が異なるため，適用対象のプログラムに適しているかどうかを判断するには，アルゴリズムの動作や性質に関する知識が要求される．また，ツールの設計や実装に依存したプログラミング上の制約も存在する．Soot の実行はパックやフェーズと呼ばれる単位に分割されており，それらの実行順序が定められているため，利用者はこのような特徴を理解し，それに従ったプログラムを記述する必要がある．また，Soot はシングルトンパターンで設計されているため，マルチスレッドによる解析や複数プログラムの同時解析には不向きである．

Soot でコールグラフを取得する場合は，たとえば図 1 のようなプログラムを記述し，以下のようにプログラムを実行する．このコマンドは，target.jar を解析対象に指定し，コールグラフを取得するためのアルゴリズムとして Class Hierarchy Analysis [3] を指定し，target.Main から参照されるクラス群を解析範囲に指定するものである．

```
java SootCallGraph -cp target.jar -whole-program -p -cg.cha -app target.Main
```

図 1 の 3 行目から 15 行目は，Soot のフレームワークに従ったパックとフェーズの記述である．プログラム全体の解析を行う “wjtp” パックに，コールグラフを取得する処理を定義した SceneTransformer オブジェクトを “wjtp.myTrans” フェーズという名前で登録している．16 行目で Soot による解析を起動することで，登録したフェーズが解析中に実行され，呼び出し関係を表す文字列が出力される．

WALA でコールグラフを取得する場合は，図 2 のようなプログラムを記述し，以下のようにプログラムを実行する．

```
java WalaCallGraph target.jar
```

図 2 の 3 から 7 行目で解析対象の指定や解析オプションの指定を行い，8 行目でコールグラフを表現したオブジェクト (CallGraph クラス) を取得する．しかし，このコールグラフは 1 つのメソッドが 1 つの頂点に対応するとは限らず， 異なる実行コンテキスト (通常はコールスタックの状態) ごとに独立した頂点が作成されるという設計になっている．そのため，1 メソッドが 1 頂点に対応するコールグラフを取得したい場合は，その旨をオプションで指定する必要があ

†1 http://asm.ow2.org/
†2 https://soba.osdn.jp/

```
 1: public class SootCallGraph {
 2:   public static void main(String[] args) {
 3:     PackManager.v().getPack("wjtp").add(new Transform("wjtp.myTrans",
                                                  new SceneTransformer() {
 4:       @Override
 5:       protected void internalTransform(String phaseName, Map options) {
 6:         CallGraph cg = Scene.v().getCallGraph();
 7:         for (Iterator<MethodOrMethodContext> callers = cg.sourceMethods();
                                                      callers.hasNext();) {
 8:           MethodOrMethodContext caller = callers.next();
 9:           for (Iterator<Edge> edges = cg.edgesOutOf(caller); edges.hasNext();) {
10:             Edge edge = edges.next();
12:             SootMethod srcMethod = edge.getSrc().method();
13:             SootMethod tgtMethod = edge.getTgt().method();
14:             System.out.println(srcMethod.toString() + " may call "
                                                    + tgtMethod.toString());
15:     }}}}));
16:     soot.Main.main(args);
17: }}
```

図 1　Soot の利用例: メソッドの呼び出し関係を抽出するプログラム (http://www.brics.dk/SootGuide/ のサンプルプログラムを引用し，すべての呼び出し関係を取得するように変更している.)

る．WALA のこのような設計は，複数のアルゴリズムで構築することができるコールグラフの表現を抽象化するためのものであるが，利用者はアルゴリズムによる解析内容の違いを認識していないとその結果を理解することができないという弱点がある．

これまで述べたような Soot や WALA の特徴は，解析アルゴリズムを良く知っている研究者が，様々なアルゴリズムを切り替えて性能評価を行うことに適した設計になっていることに由来する．一方で，ソフトウェア工学においては，常にそのような解析を行いたいわけではない．たとえばプログラムに含まれるクラスやメソッドの一覧，コールグラフといったプログラムに関する基本的な情報を各人の研究のために取得したいという要求に対しては，詳細なアルゴリズムの選択肢は不要であり，単純に対象プログラムを入力として与えれば必要な情報を get メソッドによって取得できるという設計のほうがシンプルで利用しやすいと考えられる．我々は，このような要求に適したツールとして，SOBA の設計と開発を行った．

SOBA の実装は Java バイトコードの解析ライブラリである ASM を基盤としている．ASM にはバイトコードから 1 つのクラスとして定義されたメソッドやフィールドの情報を取得する機能があるが，SOBA は JAR ファイルやディレクトリからプログラム全体を読み込む簡潔な方法を追加し，クラスの一覧やクラスの継承関係といったクラス単体の解析結果をまとめて管理する方法を提供している．また，ASM にはメソッド内の制御フロー，データフロー情報を取得する機能があるが，SOBA はそれらの機能を実行した結果を制御フローグラフ，制御依存関係，データ依存関係のオブジェクト表現へと変換して利用者に提供する．

2.2　その他の解析ツール

Java ではコンパイル後のバイトコードにも多くの情報が含まれているためにバイトコードを直接解析

```
 1: public class WalaCallGraph {
 2:   public static void main(String[] args) {
 3:     AnalysisScope scope = AnalysisScopeReader
                                  .makeJavaBinaryAnalysisScope(args[0], null);
 4:     ClassHierarchy cha = ClassHierarchy.make(scope);
 5:     Iterable<Entrypoint> entrypoints = Util.makeMainEntrypoints(scope, cha);
 6:     AnalysisOptions options = new AnalysisOptions(scope, entrypoints);
 7:     CallGraphBuilder builder = Util
                .makeZeroCFABuilder(options, new AnalysisCache(), cha, scope);
 8:     CallGraph cg = builder.makeCallGraph(options, null);
 9:     for (CGNode caller: cg) {
10:       for (Iterator<CGNode> callees = cg.getSuccNodes(caller); callees.hasNext();) {
11:         CGNode callee = callees.next();
12:         IMethod callerMethod = caller.getMethod();
13:         IMethod calleeMethod = callee.getMethod();
14:         System.out.println(callerMethod.toString() + " may call "
                                                    + calleeMethod.toString());
15: }}}}
```

図 2 WALA の利用例：メソッドの呼び出し関係を抽出するプログラム (com.ibm.wala.core.tests プロジェクトの com.ibm.wala.examples.drivers.PDGCallGraph クラスより引用し，すべての呼び出し関係を取得する処理を追加している.)

することがあるが，計算機ごとに異なるバイナリにコンパイルされるような他のプログラミング言語を対象とした場合は，ソースコードの解析が一般的である．ソースコードは，プログラミング言語ごとの文法をもとに構文木を構築する処理までは言語を問わずほぼ共通であるため，構文解析の結果として抽出されるデータ構造を共通化することで複数のプログラミング言語に適用可能なソースコード解析ツールが開発されている [1] [9] [11].

MOOSE はソフトウェアメトリクスの計測や可視化機能を提供するプラットフォームである [1]．異なる言語で書かれたソースコードの解析結果を共通のモデルで表現することによって多言語対応を実現しており，ソースコードの行数や関数の数といったように，手続き型言語に存在する主要なメトリクスを計算できる．また，提供されている可視化方式に対して入力するメトリクスを選択することで，目的に合わせた多様な可視化を行うことができる．

MASU はオブジェクト指向プログラミング言語を対象として，ソースコードから言語非依存な抽象構文木を構築することによって，統一的なメトリクスの計測を提供するプラットフォームである [9]．MASU の利用者は，メトリクスの計測ロジックをプラグインとして実装することで，様々な言語に対して独自のメトリクスを計測することができる．

UNICOEN は，プログラム解析ツールを様々な言語に対応させるためのフレームワークである [11]．バグ検出やメトリクス計測などの解析ツールを UNICOEN が提供する API を利用して開発することで，様々な言語に対してそれらのツールを適用することができる．

3 SOBA

SOBA は，Java バイトコードのプログラム解析を Java で記述するためのクラスライブラリである．SOBA を利用することで，プログラムから以下のよ

うな情報を取得することができる．

- プログラムに含まれるクラス，メソッドの一覧
- メソッドの呼び出し関係
- メソッド内の制御フロー，制御依存関係，データ依存関係

SOBA は，プログラム解析に関する詳細な知識を持たない人やプログラム解析を初めて行う人が上記の情報に容易にアクセスできるようにすることを目的としている．これらの情報は，我々の研究グループで行ってきた研究においてしばしば必要としたものである [4] [5] [6] [8].

SOBA が提供する機能は，Java プログラムの解析における基本的なアルゴリズムの実装に限られている．メソッドの呼び出し関係を取得する手法として提供している Class Hierarchy Analysis [3] は，Java の言語仕様に従って動的束縛を解決するアルゴリズムである．メソッド内の制御フロー，制御依存関係，データ依存関係を取得するアルゴリズムは，コンパイラ最適化技術として確立されている [10]．これらの基本的なアルゴリズムの実装は，低い学習コストで利用できることが望ましいと考え，SOBA は Soot や WALA に比べて規模を小さくしている (4 章にて詳述する)．メソッド間のフロー解析，依存関係解析，ポインタ解析などの技術は，現在も研究が盛んに行われており，これらの技術を利用するには，各アルゴリズムの特性を理解し，適切なアルゴリズムとパラメータを選択する必要があるため，SOBA の機能としては提供しない．SOBA を利用したうえで，これらの高度な解析が必要になった場合は，SOBA を基盤としてアルゴリズムを実装したり，Soot や WALA などの高機能なツールを利用したりして，利用者が深く学習していくことを期待している．

3.1 実装の特徴

SOBA では，利用者が解析対象の jar/zip/class ファイルやそれらを含むディレクトリを指定するだけで，対象プログラムの解析が行われる．ファイルの読み込み時には，zip ファイルは再帰的に展開されるため，複数のプログラムが含まれている zip ファイルをまとめて読み込むことができる．

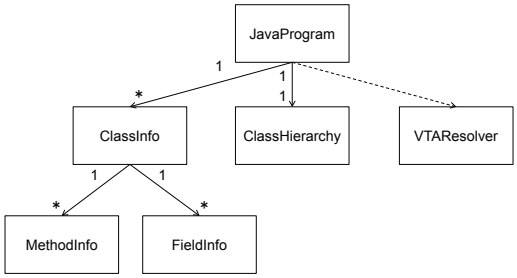

図 3　SOBA のクラス図

SOBA にプログラムを読み込ませると，図 3 に示す階層的なデータ構造 (VTAResolver を除く) が構築される．VTAResolver は呼び出し関係を取得するためのクラスであるが，多くの時間とメモリを必要とするため，利用者の要求によってそのインスタンスを生成する．利用者は各クラスが持つ get メソッドを呼び出すことで必要な情報を取得することができる．表 1 は，SOBA の主なクラスとそのクラスから取得できる情報を示したものである．制御フローグラフやデータ依存関係は，対応する get メソッドを呼び出すと計算され，MethodInfo オブジェクトに保持される．保持されたデータは解放されないため，メモリを節約するプログラムは記述しにくいが，要求された情報の分だけ実行時間を費やす設計になっている．利用者は，SOBA が直接提供する情報以外に，必要であれば ASM の持つデータ構造 (Tree API) にアクセスすることもできる．Tree API では，クラス，メソッド，命令などの情報が階層的に保持されており，文字列リテラルなどの詳細な情報を取得できる．

SOBA は，メソッド呼び出しの動的束縛を解決する手段として，Class Hierarchy Analysis (CHA) [3] と Variable Type Analysis (VTA) [12] の 2 つの実装を提供している．CHA はクラスの階層構造から，呼び出される可能性のあるメソッドの候補を列挙する．VTA は階層構造に加え，データフローを追跡することでレシーバオブジェクトに代入される可能性のあるデータ型を認識し，呼び出される可能性のあるメソッドの候補を列挙する．本来の VTA は，利用者が解析対象中の main メソッドを指定し，そのメソッドから到達可能なデータ型を計算するものであるが，

表 1 SOBA の構成

クラス名	説明	機能
JavaProgram	プログラム全体を表す	宣言されているクラス (ClassInfo) の一覧を取得する
ClassInfo	1 つのクラスを表す	クラス名，パッケージ名を取得する 宣言されているメソッド (MethodInfo) の一覧を取得する 宣言されているフィールド (FieldInfo) の一覧を取得する
MethodInfo	1 つのメソッドを表す	メソッドのシグネチャ，返り値の型を取得する 制御フローグラフ，制御依存関係，データ依存関係を取得する ASM Tree API にアクセスする
FieldInfo	1 つのフィールドを表す	フィールド名，型を取得する
ClassHierarchy	CHA [3] の実装	指定されたクラスの親クラス，子クラスを取得する メソッド呼び出しの動的束縛を解決する
VTAResolver	VTA [12] の実装	メソッド呼び出しの動的束縛を解決する

SOBA の実装では，どのメソッドからも呼び出されることのないメソッドには任意のデータ型が到達すると仮定するため，main メソッドを指定することなくプログラム全体の解析を行うことが可能である．

CHA はクラス階層によってメソッドの候補を計算するため，リフレクションを利用するプログラムに対しても呼び出される可能性のあるメソッドを漏れなく列挙することができ，呼び出し関係を取得する基本的な手段であるといえる．しかし，クラス階層上は複数の候補があっても，実際にはあるクラスの呼び出ししかありえないという場合もある (List インターフェースに対する呼び出しが実際には ArrayList しかありえないなど)．このような場合は，すべての呼び出し文で常に複数の候補が列挙されてしまうため，ありえない候補を排除したいという要求があると考えられる．SOBA では，大規模プログラムに対して適用することを考え，計算量がプログラムサイズに対して線形である VTA を採用している．

SOBA では，ローカル変数 (配列の要素を除く) を定義，参照する命令間，そしてオペランド・スタック [7] に対してプッシュ，ポップを行う命令間のデータ依存関係を取得できる．フィールド変数やメソッド呼び出しによる依存関係を取得するには，ポインタ解析などの高度なアルゴリズムが必要になるため，SOBA 自体の機能としては提供しない．

SOBA では，プログラムの読み込み時に，対象プログラムとそれが利用するライブラリを分けて指定すると，両者を識別するためのラベルが付けられる．ライブラリクラス自体の情報や，ライブラリ内での呼び出し関係には興味がないという場合に，ラベルによって識別することを想定している．

3.2 プログラム例

3.2.1 メソッドの呼び出し関係の抽出

図 4 は，メソッドの呼び出し関係を抽出するプログラムである．このプログラムは，コマンドライン引数で解析対象のファイルやディレクトリが指定されると，対象プログラム中の各メソッドがどのメソッドを呼び出すかを文字列で出力する．3 行目で，解析対象の JavaProgram オブジェクトを作成し，動的束縛を解決するために ClassHierarchy オブジェクトを 4 行目で取得している．5 行目以降のループ構造で，対象プログラム中の各クラス (ClassInfo)，クラス内の各メソッド (MethodInfo) を訪問し，7 行目ではメソッド呼び出し命令の一覧を取得している．8 行目で，各メソッド呼び出しの呼び出し先を CHA によって解決し，10 行目からのループ構造でその呼び出し先を文字列として出力している．呼び出し先のメソッドが解析対象に含まれていない場合は，解決結果が空になるため，呼び出し元の命令を文字列として出力している (14 行目)．このようなプログラムを記述することによって，コールグラフに相当する情報が得られる．

図 5 は，SOBA 自体を対象に図 4 のプログラムを実行した結果の一部を示したものである．図 4 の main メソッドから呼び出されるメソッドが，“クラス名. メソッド名 (引数): 返り値の型” という形式で列挙されている．`<init>` はコンストラクタの呼び出しであることを意味している．拡張 for 文では Iterator

```
 1: public class SobaCallGraph {
 2:   public static void main(String[] args) {
 3:     JavaProgram program = new JavaProgram(ClasspathUtil.getClassList(args));
 4:     ClassHierarchy ch = program.getClassHierarchy();
 5:     for (ClassInfo c: program.getClasses()) {
 6:       for (MethodInfo m: c.getMethods()) {
 7:         for (CallSite cs: m.getCallSites()) {
 8:           MethodInfo[] callees = ch.resolveCall(cs);
 9:           if (callees.length > 0) {
10:             for (MethodInfo callee: callees) {
11:               System.out.println(" [inside] " + m.toLongString()
                                  + " may call " + callee.toLongString());
12:             }
13:           } else {
14:             System.out.println(" [outside] " + cs.toString());
15:  } } } } } }
```

図 4　**SOBA** の利用例: メソッドの呼び出し関係を抽出するプログラム

```
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/util/files/ClasspathUtil.getClassList(java/lang/String[]:files):
    soba/util/files/IClassList[]
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/core/JavaProgram.<init>(soba/core/JavaProgram:this,
    soba/util/files/IClassList[]:lists): void
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/core/JavaProgram.getClassHierarchy(soba/core/JavaProgram:this):
    soba/core/ClassHierarchy
[inside] demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
  may call soba/core/JavaProgram.getClasses(soba/core/JavaProgram:this): java/util/List
[outside] java/util/List.iterator()Ljava/util/Iterator; called by
  demo/soba/ClassHierarchyPerformance.main(java/lang/String[]:args): void
                                    ...
```

図 5　図 4 のプログラムを実行した場合の出力結果

のメソッドが呼び出されるが，その呼び出し先はこの例の解析対象である SOBA に含まれていないため，[outside] と出力されている．

3.2.2　データ依存関係の抽出

図 6 は，メソッド内の命令間のデータ依存関係を抽出するプログラムである．6 行目まではメソッド呼び出し関係の抽出の場合と同様に各メソッドに対して処理を行うための命令の記述であり，7 行目でデータ依存関係を DataDependence オブジェクトとして取得している．そして，ループ構造で，ある命令が格納したデータを別の命令が使用するという依存関係を出力している．

```
 1:  public class DumpDataFlowEdge {
 2:    public static void main(String[] args) {
 3:      JavaProgram program = new JavaProgram(ClasspathUtil.getClassList(args));
 4:      for (ClassInfo c: program.getClasses()) {
 5:        for (MethodInfo m: c.getMethods()) {
 6:          System.out.println(m.toLongString());
 7:          DataDependence dd = m.getDataDependence();
 8:          for (DataFlowEdge e: dd.getEdges()) {
 9:            System.out.println(e.toString());
10:  } } } } }
```

図 6 SOBA の利用例: データ依存関係を抽出するプログラム

```
soba/example/dump/DumpDataFlowEdge
  .main(java/lang/String[]:args): void
PARAM -> 4 (LOCAL:0)
4 -> 5 (STACK:2)
2 -> 6 [1/2] (STACK:1)
5 -> 6 [2/2] (STACK:2)
2 -> 7 (STACK:0)
7 -> 10 (LOCAL:1)
  ...
```

図 7 図 6 のプログラムを実行した場合の出力結果

図 7 は，図 6 のプログラムを自分自身を対象に実行した結果の一部を示したものである．`i -> j` という記述は，i 番目の命令から j 番目の命令にデータ依存関係が存在することを表しており，ここでの番号は ASM におけるバイトコード命令リスト (InsnList クラス) から命令を取り出すときの get メソッドの引数に対応している．また，オペランド・スタック上の依存関係は `STACK`，ローカル変数の依存関係は `LOCAL` として出力される．この情報は，ある 1 つのプログラム文を実現する一連のバイトコード命令の中で生じるデータ依存関係と，ローカル変数を経由したプログラム文間でのデータフローとを区別するために使用できる．解析対象のバイトコードにデバッグ情報が埋め込まれている場合は，ローカル変数の名前や型，バイトコード命令に対応する行番号などの情報も取得できる．

3.3 研究における利用例

Kashima らは，SOBA を基盤としてメソッド間の依存関係解析を実装し，メソッドをまたがったデータ依存関係や制御依存関係の可視化を行っている [4]．また，Kashima らはポインタ解析を実装し，プログラムスライシングの実装を行っている [5]．これらの研究は Soot や WALA が対象とした研究の範囲に近く，これらの利用も検討したが，実行時の性能を細かく計測するために，SOBA をベースにして単純な解析アルゴリズムだけを再実装することで実現を行った．

松村らは，実行トレースの内容を用いて Java のあるメソッドのローカル変数の状態を再現するツールを実現した [8]．この実現においては，対象プログラムの読み込み，プログラムが実行したバイトコード命令に対応する ASM のデータ構造へのアクセス，そしてメソッドごとの制御フローグラフの取得に SOBA を利用した．このツールの実装は著者にとって初めて Java バイトコード情報を使用する機会であったが，様々なプログラム解析の論文を読むこともなく，情報の取得処理を短期間で実装することが可能であった．

Kashiwabara らは，各メソッドで呼び出されるメソッドのシグネチャやアクセスされるフィールドの名前や型をもとにして，メソッドの内容に相応しいメソッドの動詞を推薦する手法を提案した [6]．この研究は一種のルールマイニングであり，解析対象プログラム群からメソッド一覧を取得し，各メソッドの内容を収集する処理を SOBA によって実装した．第 1 著

表 2　行数，クラス数，コマンドラインオプションの比較

プログラム	比較項目	SOBA	Soot	WALA
呼び出し関係	行数	18	18	17
	インポート数	6	8	12
	コマンドラインオプションによる指定	解析対象プログラム	解析対象プログラム，CHA の利用，解析対象メインクラス	解析対象プログラム
データ依存関係 制御依存関係	行数	17	15	27
	インポート数	7	9	21
	コマンドラインオプションによる指定	解析対象プログラム	解析対象プログラム	解析対象プログラム
ツールの総クラス数		116	3,248	1,575

者は Soot が提供するような解析アルゴリズムに関する知識はまったく持っておらず，バイトコードに関する知識も特になかったが，単純に get メソッドでアクセスできる範囲の情報から必要な処理を短期間で記述することが可能であった．

4　Soot，WALA との比較

4.1　プログラムの記述

SOBA，Soot，WALA のそれぞれを使用した場合の記述量の比較を行った．表 2 は，CHA によって呼び出し関係を取得するプログラムとデータ・制御依存関係を取得するプログラムを記述した場合の行数とインポートが必要なクラス数を示したものである．各プログラムは 4.3 節で用いた実行性能計測用のプログラムであり，そのソースコードを OSDN にて公開している．行数は，クラス宣言の開始から終了までのうち空行と計測用コードを除いた行数である．ループ処理については，Iterable インターフェースを実装しているクラスは拡張 for 文を利用し，そうでないクラスは通常の for 文を利用して記述している．インポート数は，各ツールのパッケージ内のインポートが必要なクラス数である．ツールの総クラス数は，各ツールのパッケージ内のクラスのうち，テストクラス (名前が “Test” で終わるクラス) でないものの数である．ただし，WALA については，計測用プログラムの記述に必要なプロジェクト †3 のみを対象とした数値である．

行数については，データ・制御依存関係を取得するプログラムにおいて WALA がやや多いが，その他はあまり変わらない．インポートが必要なクラス数は，WALA が多く SOBA が少ないため，SOBA は学習が必要なクラス数が少ないといえる．SOBA は機能を限定しているため，ツール全体のクラス数も少ない．比較に用いた Soot のプログラムは，呼び出し関係の解析アルゴリズムをプログラム中で指定していないので，コマンドラインオプションで指定する．また，Soot を利用する場合は，利用者が目的の処理を記述するためにパックとフェーズの概念 †4 を学習する必要がある．

4.2　機能

表 3 は各ツールの持つ機能とその利用方法を示したものである．解析対象の指定，呼び出し関係，データ・制御依存関係の解析は，それぞれのツールで利用方法は異なるが，対応する機能を持っている．一方，ポインタ解析やメソッド間の依存関係解析については，SOBA よりも Soot や WALA のほうが優れている．ただし，メソッド間の解析は，実行コンテキストを考慮しない (1 つのメソッドのすべての呼び出しを同一に扱う) 場合，SOBA の持つ機能を利用することで実装が可能である．

Soot や WALA では指定が必要であるものが，SOBA では不要になるように設計している機能がある．呼び出し関係の解析では，Soot や WALA は解析対象のメインクラスを指定する必要があるが，SOBA では指定しない．メインクラスを指定すると，そのク

†3　com.ibm.wala.core，com.ibm.wala.shrike，com.ibm.wala.util

†4　https://github.com/Sable/soot/wiki/Packs-and-phases-in-Soot

表 3 各ツールの機能と記述方法の比較

機能	SOBA	Soot	WALA
データ構造	ASM Tree API	Jimple	Statement オブジェクト
解析対象ファイルの読み込み	ClasspathUtil.getClassList に jar/zip/class ファイルを指定	コマンドラインオプション -cp で jar/class ファイルを指定	AnalysisScopeReader. makeJavaBinaryAnalysisScope に jar/class ファイルを指定
呼び出し関係の解析	ClassHierarchy クラスのオブジェクトを取得	コマンドラインオプションで -whole-program -p cg.cha とメインクラスを指定し，Scene.v(). getCallGraph() メソッドを呼び出す	AnalysisOptions クラスのオブジェクトにメインクラスを指定し，CallGraph インターフェースを実装しているクラスのオブジェクトを取得
データ・制御依存関係の解析	DataDependence クラスのオブジェクトを取得	HashMutablePDG クラスのオブジェクトを生成	取得する依存関係を指定 (DataDependenceOptions クラス，ControlDependenceOptions クラス) し，PDG クラスのオブジェクトを生成
ポインタ解析	なし	コマンドラインオプションで -p cg.spark あるいは -p cg.paddle を指定し，Scene.v(). getPointsToAnalysis() メソッドを呼び出す	PointerAnalysis インターフェースを実装しているクラスのオブジェクトを取得
メソッド間の依存関係解析	なし	HashMutablePDG クラスやポインタ解析結果を利用	SDG クラスのオブジェクトを生成

ラスから到達可能な範囲のみに限定されて解析が行われるが，メインクラスの指定が困難な Web アプリケーションなどには不向きである．SOBA は，到達可能性に関係なくすべてのクラスを解析する設計になっているため，利用者にとって不要なクラスも自動的に解析されている場合があるが，軽量な実装になっているため，実行時間やメモリ消費量の点で不利にならないと考えられる．データ・制御依存関係の解析では，WALA はローカル変数の依存関係のみ取得するか，ヒープ領域の依存関係も取得するかなどを指定する必要があるが，SOBA では指定しない．WALA では，高度な依存関係解析を選択することができるが，SOBA では 3 章で述べたように，基本的なアルゴリズムの実装のみを提供している．

4.3 実行性能

SOBA，Soot，WALA のそれぞれを利用して，CHA によって呼び出し関係を取得するプログラムとデータ・制御依存関係を取得するプログラムを作成し，DaCapo benchmark(バージョン 9.12) [2] から選定したプログラムを対象に，各プログラムの実行時間と使用メモリを計測した．本論文では，それぞれのツールが提供するオブジェクトで目的の情報を取得するまでを計測の範囲とした．呼び出し関係を取得するプログラムの場合，SOBA では MethodInfo オブジェクト間，Soot では SootMethod オブジェクト間，WALA では IMethod インターフェースを実装したオブジェクト間の関係を取得する処理をそれぞれ計測する．データ・制御依存関係を取得するプログラムの場合，SOBA では ASM の Tree API で表現されるバイトコード命令間の依存関係，Soot では バイトコード変換した 3 番地コード (Jimple) を抽象化した PDGNode オブジェクト間での依存関係，WALA ではバイトコード命令を独自に抽象化した Statement オブジェクト間の依存関係を取得する処理をそれぞれ計測する．

表 4 にそれぞれの計測結果を示す．表 4 の値は，Intel Xeon E5-2690 2.90GHz で 10 回実行した結果の平均値である．本計測では，JDK1.8.0 をライブラリとして用いている．呼び出し関係の解析では，SOBA は Soot や WALA より高速であり，使用メモリは，WALA とは対象プログラムによって異なる

表 4　実行性能の比較 (時間はミリ秒，メモリは MB)

		呼び出し関係						データ・制御依存関係					
		SOBA		Soot		WALA		SOBA		Soot		WALA	
プログラム	メソッド数	時間	メモリ	時間	メモリ	時間	メモリ	時間	メモリ	時間	メモリ	時間	メモリ
sunflow	4,828	1,894	285	10,744	1,915	2,250	151	1,527	488	6,273	2,012	15,721	1,661
avrora	10,073	2,085	344	12,005	1,068	2,965	540	1,580	327	6,130	1,296	27,628	2,543
pmd	19,523	2,630	455	20,868	654	4,405	362	4,224	784	17,000	4,246	216,461	2,877
h2	19,962	2,508	570	35,368	5,475	5,041	1,172	4,753	1,781	53,830	7,802	143,269	6,968
batik	36,063	3,522	801	35,369	8,006	8,041	637	6,438	1,000	17,601	1,954	953,340	5,937

表 5　Eclipse 4.2 と JDK 1.7.0 を対象にした適用結果

アルゴリズム	CHA			VTA
ツール	SOBA	Soot	WALA	SOBA
実行時間	15.5 分	65.8 分	27.3 分	95.5 分
使用メモリ量	2.0GB	23.7GB	16.4GB	37.2GB

が，Soot より少ない．データ・制御依存関係の解析では，SOBA は Soot や WALA より実行時間，使用メモリともに少ない．機能を限定したこと，また軽量なライブラリである ASM を利用したことにより，SOBA は高い実行性能を実現できている．

表 5 は，Eclipse IDE for Java Developers 4.2 と Oracle JDK 1.7.0 の合計 67,973 クラス，543,425 メソッドを対象に呼び出し関係解析を行った結果である．JDK 1.8.0 を対象にすると Soot による解析がクラッシュしてしまったため，本計測では規模の小さい JDK 1.7.0 を用いている．VTA による解析については，Soot ではクラッシュしてしまい，WALA は VTA を提供しないため，SOBA の結果のみを示している．Soot や WALA は，すべての呼び出し関係をグラフオブジェクトとして保持するが，SOBA はグラフを保持しないため，メモリ使用量を抑えることができる．VTA による解析では，特に多くのメモリを必要とするが，大規模プログラムの高速な解析が可能である．

5　まとめと今後の課題

本論文では，Java バイトコード解析ツールキット SOBA について紹介した．SOBA は，プログラム解析に関する詳細な知識を持たない人でも，プログラムの読み込みや解析情報の取得を容易に行えるように設計されている．SOBA は，我々の研究グループで使われており，様々な解析の基盤として活用できるものであると考えているが，ソースコードを公開して間もないため，今後，利用者からのフィードバックを受けて発展させることが必要である．特に，SOBA の提供機能は基本的なアルゴリズムの実装に限定しているが，利用者にとって，これらの機能で十分かどうかを確認する必要がある．

謝辞 情報処理学会ソフトウェア工学研究会主催のソフトウェアエンジニアリングシンポジウム 2015 において，SOBA に対するコメントをいただいた方々に感謝します．本研究は JSPS 科研費 JP26280021 の助成を受けたものです．

参 考 文 献

[1] Baroni, A. L. and Abreu, O. B. E.: An OCL-based formalization of the MOOSE metric suite, in *Proceedings of the 7th ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, 2003.

[2] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D. and Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, in *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ACM, 2006, pp. 169–190.

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

[4] Kashima, Y., Ishio, T., Etsuda, S. and Inoue, K.: Variable Data-Flow Graph for Lightweight Program Slicing and Visualization, *IEICE Transactions on*

Information and Systems, Vol. E98-D, No. 6(2015), pp. 1194–1205.
[5] Kashima, Y., Ishio, T. and Inoue, K.: Comparison of Backward Slicing Techniques for Java, *IEICE Transactions on Information and Systems*, Vol. E98-D, No. 1(2015), pp. 119–130.
[6] Kashiwabara, Y., Ishio, T. and Inoue, K.: Improvement in Method Verb Recommendation Technique using Association Rule Mining, *IEICE Transactions on Information and Systems*, Vol. E98-D, No. 11(2015), pp. 1982–1985.
[7] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A.: The Java Virtual Machine Specification Java SE 8 Edition, https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.
[8] 松村俊徳, 石尾隆, 鹿島悠, 井上克郎: REMViewer: 複数回実行された Java メソッドの実行可視化ツール, コンピュータソフトウェア, Vol. 32(2015), pp. 137–148.
[9] 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(2009), pp. 1518–1531.
[10] 中田育男: コンパイラの構成と最適化, 朝倉書店, 2 edition, 1999.
[11] 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(2013), pp. 1234–1249.
[12] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E. and Godin, C.: Practical Virtual Method Call Resolution for Java, in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ACM, 2000, pp. 264–280.
[13] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot - a Java Bytecode Optimization Framework, in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, pp. 13–23.
[14] WALA Project: WALA: T. J. Watson Libraries for Analysis, http://wala.sourceforge.net/wiki/index.php/Main_Page.

秦 野 智 臣

2015 年大阪大学大学院情報科学研究科博士前期課程修了．同年大阪大学大学院情報科学研究科博士後期課程入学．プログラムの静的解析，プログラム理解に関する研究に従事．

石 尾 隆

2003 年大阪大学大学院基礎工学研究科博士前期課程修了．2006 年同大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員 (PD)．2007 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教．博士 (情報科学)．プログラム解析，ソフトウェア再利用に関する研究に従事．

井 上 克 郎

1956 年生．1979 年大阪大学基礎工学部情報工学科卒業．1984 年同大学大学院博士課程修了．同年同大学基礎工学部助手．1984–1986 年ハワイ大マノア校情報工学科助教授．1989 年大阪大学基礎工学部講師．1991 年同助教授．1995 年同教授．2002 年大阪大学大学院情報科学研究科教授．2012 年大阪大学大学院情報科学研究科・研究科長．工学博士．ソフトウェア工学，特に，ソフトウェア開発手法，プログラム解析，再利用技術の研究に従事．