# Java プログラムの動的解析のためのトレーサ埋め込みツール
# Injecting Tracers into Java Class Files for Dynamic Analysis

**玉田 春昭　　門田 暁人　　中村 匡秀　　松本 健一**

奈良先端科学技術大学院大学 情報科学研究科
〒 630-0101 奈良県生駒市高山町 8916–5
E-mail: {harua-t, akito-m, masa-n, matumoto}@is.naist.jp

## 概要

本稿では，Java プログラムの耐タンパ性を評価するツール AddTracer を提案する．このツールは Java を対象とし，プログラム実行中の変数の値やメソッド呼び出しの関係を出力するトレーサを直接クラスファイルに埋め込む．そのため，ソースコードが提供されていないプログラムに対しても適用可能であり，ソースコードの構文規規則にとらわれることなく，プログラム中の任意の箇所にトレーサを挿入することができる．AddTracer を用いてソフトウェアに含まれる秘密情報 (暗号鍵など) の機密性を調べることで，ソフトウェアの改ざんや再利用の困難さ (耐タンパー性) を評価することができる．

**キーワード** 動的解析, トレーサ, Java クラスファイル

## 1　はじめに

プログラム解析の強力な一手法として，実行時のプログラムの挙動を調べる動的解析がある．そのためのツールとして，多くのプログラミング環境にはデバッガが用意されているが，利用できるデバッガの種類はプラットフォームに依存する，プログラムのソースコードを必要とする，初心者には敷居が高いといった問題があり，デバッガを利用した動的解析が常に効果的であるとは限らない．

本研究では，動的解析を支援するために，任意の Java クラスファイルに**トレーサ** (プログラムの状態をモニタリングするコード) を自動挿入するツール，AddTracer を開発した [5]．トレーサは Java プログラム実行時に参照/代入された任意の変数の状態 (値) やメソッドの開始や終了を出力する．AddTracer は，クラスファイルに直接トレーサを挿入するため，ソースコードを必要としない．また，バイトコードに処理を施すため，ソースコードの構文規則に囚われることなく，プログラム中の任意の箇所にトレーサが挿入できる．従って，任意の時点でのプログラムの実行状態を出力できる．

AddTracer は様々な用途に使用できる．まずプログラムのセキュリティに用いることができる．例えば，DVD 再生ソフトウェアのような暗号鍵を含むプログラムにおいて，生の鍵データがメモリ上に現れないことを確認することができる．AddTracer では，プログラム実行時に変数に値が代入されたとき，その値を表示するようトレーサコードを埋め込むことができる．特定の変数に鍵データが生で代入されるような脆弱なプログラムに AddTracer を適用すると，鍵データが出力されることになる．また，AddTracer を用いることで，たとえ初心者であっても容易にプログラムの動的解析を行うことができ，セキュリティ評価を行うことが可能となる．

加えて，このツールは耐タンパ性評価だけでなく，プログラムの理解性評価のための仮想メンタルシミュレーションでも用いることができる．

## 2　Java クラスファイルと Java 仮想マシン

Java クラスファイルの構造は Java 仮想マシン仕様 [2] で厳密に定められており，全ての Java クラスファイ

| 上位構造 | 属性構造 |
|---|---|
| ClassFile | SourceFile_attribute |
| | InnerClasses_attribute |
| | Deprecated_attribute |
| | Synthetic_attribute |
| field_info | ConstantValue_attribute |
| | Synthetic_attribute |
| | Deprecated_attribute |
| method_info | Code_attribute |
| | Exceptions_attribute |
| | Synthetic_attribute |
| | Deprecated_attribute |
| Code_attribute | LineNumberTable_attribute |
| | LocalVariableTable_attribute |

表 1　構造と取り得るべき属性

ルはそのフォーマットに従うことになっている [7]．この章では AddTracer を実装するために必要な情報を述べる．また，ここで述べる Java クラスファイルのバージョンは J2SDK 1.4 の仕様に基づいており，Java 5 は対象としていない．

## 2.1　クラスファイル

Java クラスファイルはいくつかのブロックから構成され，各ブロックがさらに小さなブロックより構成され，そのブロックが構造を持っている．代表的なブロックはコンスタントプール，フィールド情報，メソッド情報，属性の 4 つである．コンスタントプールは定数やシンボル名などの集合をまとめる部分である．そして，フィールド情報とメソッド情報はその名の通り Java のフィールドとメソッドをそれぞれ表している．属性は他のブロックの下位ブロックとして扱われ，フィールドの初期値やメソッドのコードなど様々な種類が存在し，それぞれ構造が異なる．また，各属性がどの構造の下位構造になるかは表 1 のように定められている．

## 2.2　コンスタントプール

コンスタントプール (Constant_Pool) には，クラス名やフィールド名，メソッド名などのシンボル情報，クラスやインターフェース，フィールド，メソッドの参照，値の初期値のエントリが格納されている．これらを区別するにはエントリの 1 バイト目の**タグ**と呼ばれる部分を見ることで可能である．

例えば，図 1 の `"Hello World!"` という文字列は定数であるため，CONSTANT_String という形式のエントリで Constant_Pool に格納されている．この

```
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

図 1　サンプルプログラム (Hello World)

```
<method_descriptor> ::= '(' <argument_type> ')' <return_type>
<argument_type>     ::= <data_type>*
<return_type>       ::= <data_type> | 'V'
<field_descriptor>  ::= <data_type>
<data_type>   ::= <base_type> | <object_type> | <array_type>
<base_type>   ::= 'B' | 'C' | 'D' | 'F' | 'I' | 'J' | 'S' | 'Z'
<object_type> ::= 'L' <fullclassname> ';'
<array_type>  ::= '[' <data_type>
```

図 2　ディスクリプタの BNF

CONSTANT_String は CONSTANT_Utf8 という同じ Constant_Pool に格納されているエントリを参照しており，実際の Hello World! という文字列のバイトはここに格納されている．

表 2 に Constant_Pool のエントリの種類を挙げる．ここで，メソッド関連のエントリである CONSTANT_Methodref や CONSTANT_InterfaceMethodref はそのメソッドが定義されているクラスの CONSTANT_Class へのインデクスと CONSTANT_NameAndType へのインデクスを持ち，メソッドの名前，引数，返り値の型は対応する CONSTANT_NameAndType で定義される．フィールドを表すエントリである CONSTANT_Fieldref も CONSTANT_Methodref と同じようにフィールドが定義されているクラスの CONSTANT_Class へのインデクスと CONSTANT_NameAndType へのインデクスを持つ．フィールドの名前，型は対応する CONSTANT_NameAndType で定義される．この CONSTANT_NameAndType では型 (メソッドならば返り値の型と引数の型) を CONSTANT_Utf8 エントリで descriptor として指定され，その BNF 表現を図 2 に示す．ここで，`<fullclassname>` はクラスの完全修飾名を指す．また，Java 仮想マシンではプリミティブタイプを表 3 のように一文字で表す．

## 2.3　属性

クラスやフィールド，メソッドには属性 (Attribute) が定義されている場合がある．例えば static final で宣

| 名前 | タグ | 説明 |
|---|---|---|
| CONSTANT_Utf8 | 1 | Utf8 形式のユニコード文字列 |
| CONSTANT_Integer | 3 | int リテラル |
| CONSTANT_Float | 4 | float リテラル |
| CONSTANT_Long | 5 | long リテラル |
| CONSTANT_Double | 6 | double リテラル |
| CONSTANT_Class | 7 | クラスまたはインターフェースへの参照 |
| CONSTANT_String | 8 | String リテラル |
| CONSTANT_Fieldref | 9 | フィールドへの参照 |
| CONSTANT_Methodref | 10 | クラスで宣言されたメソッドへの参照 |
| CONSTANT_InterfaceMethodref | 11 | インターフェースで宣言されたメソッドへの参照 |
| CONSTANT_NameAndType | 12 | メソッド参照やフィールド参照の名前と型の部分を定義するもの |

表 2　コンスタントプールのエントリの種類

| descriptor | Java の型 | descriptor | Java の型 |
|---|---|---|---|
| B | byte | C | char |
| D | double | F | float |
| I | int | J | long |
| S | short | Z | boolean |
| V | void | | |

表 3　base_type の対応

```
public void iadd(){
    int operand1 = (int)OperandStack.pop();
    int operand2 = (int)OperandStack.pop();
    int result   = operand1 + operand2;
    OperandStack.push(result);
}
```

図 3　iadd の模式

言されたフィールドには ConstantValue 属性が定義され，Constant_Pool 内の対象フィールドの初期値を表すインデックスがその属性の中で指定される．

また，native でも abstract でもないメソッドには必ず Code 属性が付随しており，その Code 属性中にメソッドの本体であるインストラクションコードが格納されている．この Code 属性はバイトコードに対して何らかの処理を行う場合に非常に重要になる．

Code 属性には，LineNumberTable 属性, LocalVariableTable 属性が含まれる場合がある．普通にコンパイルする場合には LocalVariableTable 属性は含まれないが，デバッグオプションを加えてコンパイルすることで，これらの属性が含まれるようになる．この二つの属性はそれぞれバイトコードと行数のマッピングテーブルとローカル変数の名前を管理する属性である．これらは，変数名，参照されたときの行数を出力するために必要な属性である．

また，Code 属性には *exception_table* と呼ばれる例外ハンドラを格納する場所も存在する．この部分は Java 言語の catch を実現する箇所であり，バイトコードに対して，何か処理を加えた場合，この exception_table に含まれるプログラムカウンタを修正する必要がある．

## 2.4　オペランドスタック

クラス中のメソッドが実行されるとき，オペランドスタック (Operand Stack) と呼ばれる記憶領域が確保される．多くの命令がオペランドスタックを経由して値を受け取ったり，値を渡したりする．これはメソッドの実行ごとに生成され，メソッドの実行が終わると削除される．

2 つの int 型変数の足し算を行う命令 (インストラクション)，iadd を例に挙げる．iadd には引数が定義されておらず，オペランドスタックの上から値を 2 つ読み込んで足し算を行い，結果をオペランドスタックに積むと定義されている．iadd はバイトコード中のインストラクションであるが，意味的には図 3 に挙げたソースコードと同じ振る舞いをする[*1]．

オペランドスタックに対して 1 回のポップで 1 ワードの領域がポップされる．多くの変数は 1 ワードの領域で表すことができるが，double 型と long 型の変数のみはこれを表すために 2 ワードの領域が必要である．2 ワードの領域を必要とする値がオペランドスタックに積まれているとき，1 ワードの領域のポップを行うことはできず，2 ワードをポップする pop2 命令を使う必要が

[*1] このソースコードはオペランドスタックの説明のために定義しただけのもので，Java 仮想マシンがこのような実装をしているとは限らない．

ある．

これ以降，スタックという言葉はこのオペランドスタックを指す．

## 2.5 インストラクション

Java 仮想マシンには様々な命令があり，データ操作，算術演算，型変換，フロー制御，スタック操作，その他と 6 種類に大別できる．データ操作はスタックに対する操作やローカル変数，配列，フィールドやオブジェクトの生成などが含まれる．

算術演算は四則演算や論理演算が各型ごとに用意されている．`iadd` や `imul`，`isub` などが存在し，それぞれ足し算や掛け算，引き算の命令を表す．また，最初の一文字目は型を表しており，i が int 型に対する命令，l が long 型，f が float 型，d が double 型に対する命令になっている．ここで，Java 仮想マシンでは byte, boolean, short, char 型は全て int 型と同じように表されるため，それらに対する命令は存在しない．

型変換は Java 言語のキャストに相当するもので，プリミティブ型の変換に用いられる命令である．`i2l` や `i2f`, `i2d` などが存在し，それぞれ int 型を long 型へ，int 型を float 型へ，int 型を double 型へ変換する．

フロー制御は実行制御を移す命令として，条件分岐や比較，無条件ジャンプ，テーブルジャンプが存在する．また，メソッドの呼び出し (invoke 命令) やメソッド終了と呼び出し元への復帰命令 (return 命令) もここに含まれる．

スタックを操作する命令には dup, dup_x1, dup_x2, dup2_x1, dup2_x2, pop, pop2, swap が存在する．これらの命令を使いスタックを操作することはバイトコードを直接扱う場合には非常に重要であるため，これらの意味を表 4 に示す．

その他のインストラクションとして，例外の送出 (`athrow`) やデバッグのための命令 (`breakpoint`)，そして，synchronized ブロックのための命令 (`monitorenter`, `monitorexit`) が存在する．

## 2.6 ベリファイア

Java 仮想マシンはクラスをロードするとき，クラスファイルの正当性をチェックする．そのチェックは大きく 4 つのパスに分かれている．

パス 1 ではクラスファイルのフォーマット検査が行われる．クラスファイルの先頭 4 バイトが正しいマジックナンバーである CA FE BA BE になっているかどうかや，属性の数が 2 と書かれているならば必ず 2 個の属性が含まれているか，また，クラスファイルが完全かどうかや，クラスファイルの終わりに余計なバイトがないかどうかなどを検査する．

パス 2 ではクラスファイル内の一貫性や他のクラスファイルとの一貫性をチェックする．このパスでバイトコード部分以外のクラスファイルのチェックが終了する．このパスでは主に以下のことを検査する．

- final クラスがサブクラス化されていないか，final メソッドがオーバーライドされていないか
- クラスの直接のスーパークラスが存在するか
- コンスタントプールが正しいかどうか．コンスタントプールの長さを超えたインデックス参照がないかどうか
- コンスタントプール内のフィールド参照，メソッド参照が正しい名前，クラス，引数であるか

パス 3 は各メソッドに含まれる Code 属性からデータフロー解析により以下のことを検証する．このときの検証を特にバイトコードベリフィケーションと呼ぶ．

- どのような経路を通っても，オペランドスタックには常に同じサイズ，同じ型の値が積まれているかどうか
- 初期化されていないローカル変数へアクセスしていないかどうか
- 適切な引数でメソッドを起動しているか
- フィールドへの代入は，適切な型の値のみになっているか
- インストラクションがオペランドスタック，もしくはローカル変数にアクセスするとき，リクエストされる型と積まれている型が一致するか

パス 4 は仮想パスであり，適切な Java 仮想マシン命令によって行われるまでこのベリファイは遅らることができる．

- アクセスするメソッドもしくはフィールドが定義されているとあるクラスが存在するかどうか

| 命令 | 動作 |
|---|---|
| dup | スタックの一番上の値を複製し，複製された値をスタックの一番上に積む |
| dup_x1 | スタックの一番上の値を複製し，複製された値をスタックの上から 2 番目と 3 番目の間に挿入する |
| dup_x2 | スタックの一番上の値を複製し，複製された値をスタックの上から 3 番目と 4 番目の間に挿入する |
| dup2 | スタックの上から 2 つの値を複製し，複製された値をスタックの一番上に積む |
| dup2_x1 | スタックの上から 2 つの値を複製し，複製された値をスタックの上から 2 番目と 3 番目の間に挿入する |
| dup2_x2 | スタックの上から 2 つの値を複製し，複製された値をスタックの上から 3 番目と 4 番目の間に挿入する |
| pop | スタックの一番上の値をスタックから取り除く |
| pop2 | スタックの上から 2 つの値をスタックから取り除く |
| swap | スタックの一番上と 2 番目の値の順番を入れ替える |

表 4　スタック操作命令

- そのクラスファイルは本当にそのクラスを表したものなのかどうか
- 他メソッドもしくはフィールドが所定のディスクリプタを持っているかどうか
- 現在のメソッドからそのメソッドもしくはフィールドにアクセス可能かどうか

もし，このパス中でベリファイに失敗すれば LinkageError もしくはそのサブクラスの例外が投げられる．

# 3　AddTracer の実装

## 3.1　実装方針

AddTracer の目的は，プログラムの実行時の変数値やメソッドの呼び出しなどを効果的に取得することである．そのアプローチとして，実行時に変数が参照/更新されたとき，またはメソッドが呼ばれたときに，変数，メソッドの名前とその値を標準出力に出力するためのコード (トレーサ) をクラスファイルのバイトコードに直接埋め込む方法を採用する．これを実現する他のアプローチとして Java 仮想マシンに手を加える方法も考えられるが [6]，Java 仮想マシンのバージョンによって大きく実装を変更する必要があるため，ポータビリティが低くなると考えられる．これに対し，Java クラスファイルに直接トレーサを埋め込む方式の場合，クラスファイルのバージョンに依存するが，クラスファイルを扱うライブラリのバージョンアップにより，多くの部分が吸収され，プログラムの変更は最小限で済む．

実装は Java のバイトコードを扱うためのライブラリである BCEL [1] を用いた。

## 3.2　トレーサの埋め込み

Java ではクラスがロードされるとき，ベリファイアのチェックに通らなければクラスローディングに失敗する．そのため，このベリファイアのチェックを通るようにトレーサを挿入するように注意しなければならない．

また，トレーサで出力する内容は変数の名前，値，そのときの行数とする．変数名はローカル変数ならば，`LocalVariableTable` から取得することができる．フィールドならば，フィールドを操作するインストラクションの引数の指す Constant_Pool のインデックスから取得することができる．行数はバイトコードのオフセットより `LineNumberTable` から取得できる．

ここで一番問題となるのは，変数の値を表示することである．変数の値は，ユーザの入力により変わったり，環境によっても変わる場合があり，全ての変数の値を静的に取得することは不可能である．そのため，実行時の変数の値を得ることができるようなコードを静的に作成する必要がある．

ローカル変数やフィールドが参照されたとき，参照された値はオペランドスタックに積まれる．また，ローカル変数やフィールドへの代入が行われるとき，代入される値はオペランドスタック上に積まれている．つまり，オペランドスタック上に全ての変数の値が現れる．そのため，オペランドスタックを操作することで変数の値を動的に得ることができる．

以上に加え，配列の要素にアクセスしたときの変数名にインデックスも出力する．配列の要素にアクセスするときも，オペランドスタックにインデックスが積まれているため，オペランドスタックを操作することで実現可能である．

```
1: public class Adder{
2:     public int add(int v1, int v2){
3:         int r = v1 + v2;
4:         return r;
5:     }
6: }
```

図 4　サンプルプログラム Adder

```
Compiled from "Adder.java"
public class Adder extends java.lang.Object{
public int add(int,int);
  Code:
   0:   iload_1
   1:   iload_2
   2:   iadd
   3:   istore_3
   4:   iload_3
   5:   ireturn
}
```

図 5　Adder のバイトコード

## 3.3　実装の詳細

ここではどのようにオペランドスタックを操作するのかを例を用いて説明する．ここで，トレーサ埋め込み処理の対象となるのは処理対象となるクラスに含まれる abstract, native 宣言されていない全てのメソッドである．abstract 宣言されているメソッドは中身がなく，トレーサを挿入する必要がない．また，native 宣言されているメソッドは中身が Java のコードではないため，トレーサを埋め込むには別の技術が必要である．

### 3.3.1　メソッド呼び出し，変数のトレーサ

まずは基本として，メソッドの開始/終了トレーサと変数参照/代入に関するトレーサ挿入法について述べる．例として，図 4 のプログラムにトレーサを埋め込むこととする．このプログラムを javac でコンパイルすると図 5 のようなバイトコードが得られる[*2]．

図 5 において，iload_1, iload_2 はそれぞれローカル変数 1 番目 (v1) と 2 番目 (v2) の参照である[*3]．参照された値はオペランドスタックに積まれる．続く iadd はオペランドスタックに積まれた 2 つの整数の和をスタックに積む命令である．そして，istore_3 で 3 番目のローカル変数 (r) にスタックの一番上の値を代入する．iload_3 で 3 番目のローカル変数 (r) の値をスタックに積み，その値を ireturn でメソッドの呼び出し元に返している．この処理をオペランドスタックの状態と各命令の関係は図 6 のようになる．

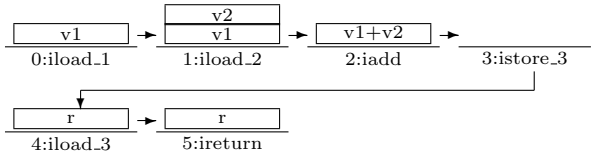


図 6　Adder#add が実行されたときのオペランドスタックの状態

```
getstatic java/lang/System#out
ldc "start Adder#add at line 3"
invokevirtual java/io/PrintStream#
            println(Ljava/lang/String;)V;
```

図 7　System.out.println のバイトコード

メソッド開始，終了トレーサで出力する関数名とその行数は，静的に解決できる．System.out.println(”start Adder#add at line 3”); に対応する命令をオフセット 0 に，そして，ireturn の直前 (図 5 においては 4 と 5 の間) に挿入することで出力できる．

メソッド開始トレーサで出力する内容はトレーサ挿入時に構築し，構築した文字列を，まず Constant_pool に挿入してから図 7 のバイトコードをバイトコードのオフセット 0 に挿入する．メソッド終了トレーサも開始トレーサと同じように構築した文字列を Constant_pool に挿入してから図 7 のようなバイトコードを ireturn の直前に挿入する．

図 7 において, getstatic 命令は指定されたクラスのスタティック変数をスタックに積む命令である．そして，ldc 命令は Constant_pool から定数値を取り出し，スタックに積む命令である，そして，最後の invokevirtual メソッドはインスタンスメソッドを実行する命令であるが，メソッドの引数と対象のインスタンスはオペランドスタックから取得する．そのため，invokevirtual の前に getstatic で実行の対象のインスタンスを，そして，実行するメソッドの引数である文字列をスタックに積んでおく必要がある．図 7 のバイトコードでは，”start Adder#add at line 3” という文字列が, 当該メソッドが実行された瞬間に，標準出力に出力されることになる．

---

[*2] このバイトコードは J2SDK 付属のツール，javap にオプション c を付けることにより得られる．また，説明に不必要な情報 (コンストラクタ) は省略している

[*3] インスタンスメソッドの場合，0 番目の変数は this を表す．スタティックメソッドの場合，this がないため，番号が一つずれる

```
getstatic java/lang/System#out
new java/lang/StringBuffer
dup
invokespecial java/lang/StringBuffer#<init>()V
ldc "v1 reference(line 3): "
invokevirtual java/lang/StringBuffer#
        append(Ljava/lang/String;)Ljava/lang/StringBuffer;
iload_1
invokevirtual java/lang/StringBuffer#
        append(I)Ljava/lang/StringBuffer;
invokevirtual java/lang/StringBuffer#
        toString()Ljava/lang/String;
invokevirtual java/io/PrintStream#
        println(Ljava/lang/String;)V;
```

図 8　System.out.println のバイトコード (文字列操作あり)

次に変数の参照に関するトレーサを挿入する．このとき，変数の値は実行時でなければ解決できない．すなわち，先ほどのメソッド開始，終了トレーサのように単純に解決できるわけではない．

そこで，まず，ソースコードレベルでの解法を考える．ソースコードであれば，変数が参照されるごとに System.out.println(”v1 reference(line 3): ” + v1); という一行を加えればよい．バイトコードでも同じコードを挿入することで解決できる．この println 文はコンパイルされると図 8 のようなバイトコードになる．

図 8 で示されるバイトコードを図 5 の 0, 1, 4 の前にそれぞれ挿入することで，変数が参照されたときのトレーサを実現できる．ただし，このバイトコード中の iload_1 は変数が参照されるバイトコードに読み替える必要がある．すなわち，図 5 において v2 の参照トレーサでは iload_2 に，r の参照トレーサでは iload_3 に読み替える必要がある．そして，図 8 の iload_1 の次の invokevirtual で呼び出している StringBuffer クラスの append メソッドの引数は参照する変数の型に合わせる必要がある．この例では，int 型変数を参照しているため，引数が I になっている．もし，参照する命令が double 型であった場合，iload ではなく，dload になり，append の引数も I ではなく，D になる．

変数への値の代入も変数参照と同じトレーサ挿入方法で実現できる．ここではローカル変数の参照/代入で説明したが，フィールドの参照/代入も 上記のバイトコードの iload の部分と append の引数の型を適切に変更することでトレーサを挿入することができる．

メソッドが開始されたとき，引数に値が代入されると考え，メソッド開始トレーサ出力後に引数に値が代入されたというトレーサを挿入する．メソッドの引数の数と型はメソッドディスクリプタから取得できる．そして，引数はローカル変数として扱われるので，ローカル変数番号の 1 から順に引数の数だけトレーサを挿入する．ただし，スタティックメソッドの場合は 0 からとなる．

```
 0: iload
 1: iload
 2: iadd
 3: dup
 4: getstatic java/lang/System#out
 5: new java/lang/StringBuffer
 6: dup
 7: invokespecial java/lang/StringBuffer#<init>()V
 8: ldc "+ operation(line 3): "
 9: invokevirtual java/lang/StringBuffer#
         append(Ljava/lang/String;)Ljava/lang/StringBuffer;
10: dup2_x1
11: pop2
12: invokevirtual java/lang/StringBuffer#
         append(I)Ljava/lang/StringBuffer;
13: invokevirtual java/lang/StringBuffer#
         toString()Ljava/lang/String;
14: invokevirtual java/io/PrintStream#
         println(Ljava/lang/String;)V;
```

図 9　演算結果出力のバイトコード

そして，図 5 において，最後に iadd が実行された直後に，演算結果と加算が実行されたというトレーサを挿入する．iadd のような演算命令はオペランドスタックから必要な数だけ値を pop し，演算結果をスタックに push する．例えば，加算を表す iadd なら 2 つの値をポップし，符号反転を実現する ineg ならば 1 つの値をポップする．さて，iadd などの計算する命令の直後にトレーサを挿入するため，演算結果はスタックの一番上に積まれている．しかし，その値をトレーサの出力で使ってしまうと本来の処理ができなくなってしまう．そのため，スタック上のエントリを複製する必要がある．しかし，複製したエントリの上に出力のための out と StringBuffer を新たに積まなければならず，その下に積まれている演算結果を一番上に持ってくる必要がある．これを解決するためにスタックを操作する．そのときのバイトコードを図 9 に，そのバイトコード実行時のスタックの状態を図 10 に示す．両図において，新たに追加するコードは iadd より後ろのコードである．

### 3.3.2　配列

配列の要素を参照したプログラムにトレーサを挿入するとき，そのインデックスも含めて変数名にする．すなわち，array という配列のインデックス 2 の要素が参照されたとき，その変数名は array だけではなく，array[2] というように表現する．

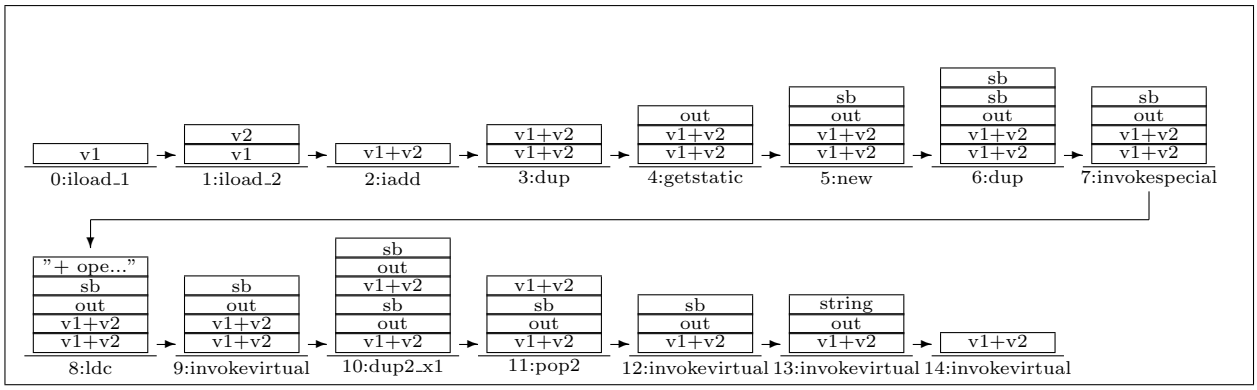


図 10　iadd へのトレーサ挿入後のオペランドスタックの状態

```
1: public class Array{
2:     public int getItem(int[] array, int index){
3:         return array[index];
4:     }
5: }
```

図 11　サンプルプログラム Array

```
Compiled from "Array.java"
public class Array extends java.lang.Object{
public Array();
 Code:
  0: aload_0
  1: invokespecial #1; //Method java/lang/Object."<init>":()V
  4: return

public int getItem(int[],int);
 Code:
  0: aload_1
  1: iload_2
  2: iaload
  3: ireturn
}
```

図 12　Array のバイトコード

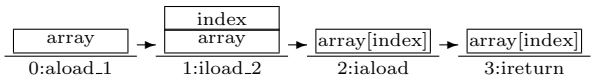


図 13　Array#getItem が実行されたときのオペランドスタックの状態

まず array[1] は iaload というインストラクションで実現される．そのインストラクションが実行されるとき，スタックには配列の参照とインデックスが積まれている．つまり，図 12 の getItem が実行されるときのオペランドスタックの状態は図 13 のようになる．

さて，配列の参照/代入に対するトレーサでは，変数名にインデックスを含め，さらにその要素の値も出力させる．インデックスは計算により求められる場合もあり，再び簡単に得られるとは限らない．例えば，副作用のあるメソッドの返り値としてインデックスが得られる場合が考えられる．この場合，再びメソッドを呼び出しても，同じ値は返ってこない．また，再びメソッドを呼び出すことでプログラムの外部仕様が変更される可能性もある．そのため，このトレーサ挿入を実現するには，配列の要素にアクセスする iaload インストラクションが実行される直前にトレーサを挿入する必要がある．

加えて，Java 仮想マシン上で配列にアクセスするためには，まず配列の参照をスタックに積み，その参照とインデックスを用いて要素にアクセスする．これは多次元配列でも同じである．例えば int 型の 2 次元配列 array の要素 array[1][2] にアクセスするには以下の手順で行われる．

1. array を push
2. 1 を push
3. array と 1 を pop し，array[1] を push
4. 2 を push
5. array[1] と 2 を pop し，array[1][2] を push

このような手順で参照されるため，変数名を array[1][2] の形にするには非常に困難である．上の例の 1, 2 が定数ならばそれほど難しくはないが，もし，変数やメソッドの返り値，もしくは計算により得られた値であるとなると，文字列の構築途中に他のトレーサが挿入されるため，プログラムが非常に煩雑になる．そこで，配列に関してはオブジェクト id と型で表される文字列を変数名として扱うことで代替案とする．すなわち，array を参照するときの変数名を `[[I@12345` のよ

```
 0: aload_1
 1: iload_2
 2: dup2
 3: swap
 4: getstatic java/lang/System#out
 5: new java/lang/StringBuffer
 6: dup
 7: invokespecial java/lang/StringBuffer#<init>()V
 8: dup2_x1
 9: pop2
10: invokevirtual java/lang/StringBuffer#
        append(Ljava/lang/Object;)Ljava/lang/StringBuffer;
11: ldc "["
12: invokevirtual java/lang/StringBuffer#
        append(Ljava/lang/String;)Ljava/lang/StringBuffer;
13: dup2_x1
14: pop2
15: invokevirtual java/lang/StringBuffer#
        append(I)Ljava/lang/StringBuffer;
16: ldc "]: "
17: invokevirtual java/lang/StringBuffer#
        append(Ljava/lang/String;)Ljava/lang/StringBuffer;
18: dup2_x2
19: pop2
20: dup2_x2
21: iaload
22: invokevirtual java/lang/StringBuffer#
        append(I)Ljava/lang/StringBuffer;
23: invokevirtual java/lang/StringBuffer#
        toString()Ljava/lang/String;
24: invokevirtual java.io.PrintStream#
        println(Ljava/lang/String;)V
25: iaload
```

図 14　iaload に対するトレーサ挿入後のバイトコード

```
dup2_x1
swap
dup2_x1
pop2
dup_x2
getstatic java/lang/System#out
new java/lang/StringBuffer
dup
invokespecial java/lang/StringBuffer#<init>()V
dup2_x1
pop2
invokevirtual java/lang/StringBuffer#
    append(Ljava/lang/Object;)Ljava/lang/StringBuffer;
ldc "["
invokevirtual java/lang/StringBuffer#
    append(Ljava/lang/String;)Ljava/lang/StringBuffer;
dup2_x1
pop2
invokevirtual java/lang/StringBuffer#
    append(I)Ljava/lang/StringBuffer;
ldc "]: "
invokevirtual java/lang/StringBuffer#
    append(Ljava/lang/String;)Ljava/lang/StringBuffer;
dup2_x1
pop2
invokevirtual java/lang/StringBuffer#
    append(<basic type of array>)Ljava/lang/StringBuffer;
invokevirtual java/lang/StringBuffer#
    toString()Ljava/lang/String;
invokevirtual java/io/PrintStream#
    println(Ljava/lang/String;)V
dup_x2
pop
```

図 16　1 ワードの場合の配列への代入

うにする．このような変数名を得るには配列変数に対して toString を呼び出せばよい．

ここでは出力形式として“`[I@12345[1]:  10`”のような形式とする．[I@12345 は array に対して toString を実行して得られる文字列であり，@ の前が型を表し，@ の後ろがオブジェクト ID を表している．そして，1 は iaload が実行されるとき，一番上に積まれている値，すなわち，参照する配列のインデックスである．最後の 10 は array[1] で得られる値であるとする．

つまり `array + "[" + 1 + "]: " + array[1]` のようなコードを実行することになる．ただし，array の参照や 1 の取得は 1 回のみで，トレーサで用いる値は dup 命令などで複製した値である必要がある．なぜなら，これはメソッド呼び出しにより得られた値かもしれないためである．

では実際に iaload 命令に対してトレーサを埋め込む．トレーサ挿入後のバイトコードは図 14 に，そのときのスタックの状態を図 15 に表す．最初の 2 命令 (iaload_1, iconst_1) と一番最後の命令 (iaload) は元々の命令である．その間の命令が埋め込んだコードである．

配列への代入はスタックに 3 つの値が積まれているため，参照よりも複雑になる．ここでは紙面の都合上，スタックに 2 ワードで積まれる double, long 型と 1 ワードで詰まれるその他の型の 2 種類のトレーサの結果のみを示す．

### 3.3.3　インクリメント

`i++` や `i+=10` で表されるインクリメントは，直感的には変数を参照し，計算を行い，その結果を代入するという命令である．バイトコードではこのインクリメント専用の命令が存在し，iinc という一つの命令でこの一連の動作を実現している．しかし，トレーサを見る側としては `i+=10;` であっても `i = i + 10;` であってもインクリメントすること変わりはない．そのため，インクリメントであっても，そうでない場合でも同じトレーサが出力されるほうが望ましい．

iinc という命令はオペランドスタックを全く使わず，ローカル変数のみを処理の対象にしている．そして，この命令はローカル変数のインデックスと増分値の二つの引数を取る．

さて，iinc (`i+=10;`) と普通の演算 (`i = i + 10;`) のトレーサを同じようにするには，iinc の前後にトレーサを挿入すればよい．つまり，iinc が実行される前に iinc が扱う変数の参照トレーサを挿入し，iinc の実行後に演

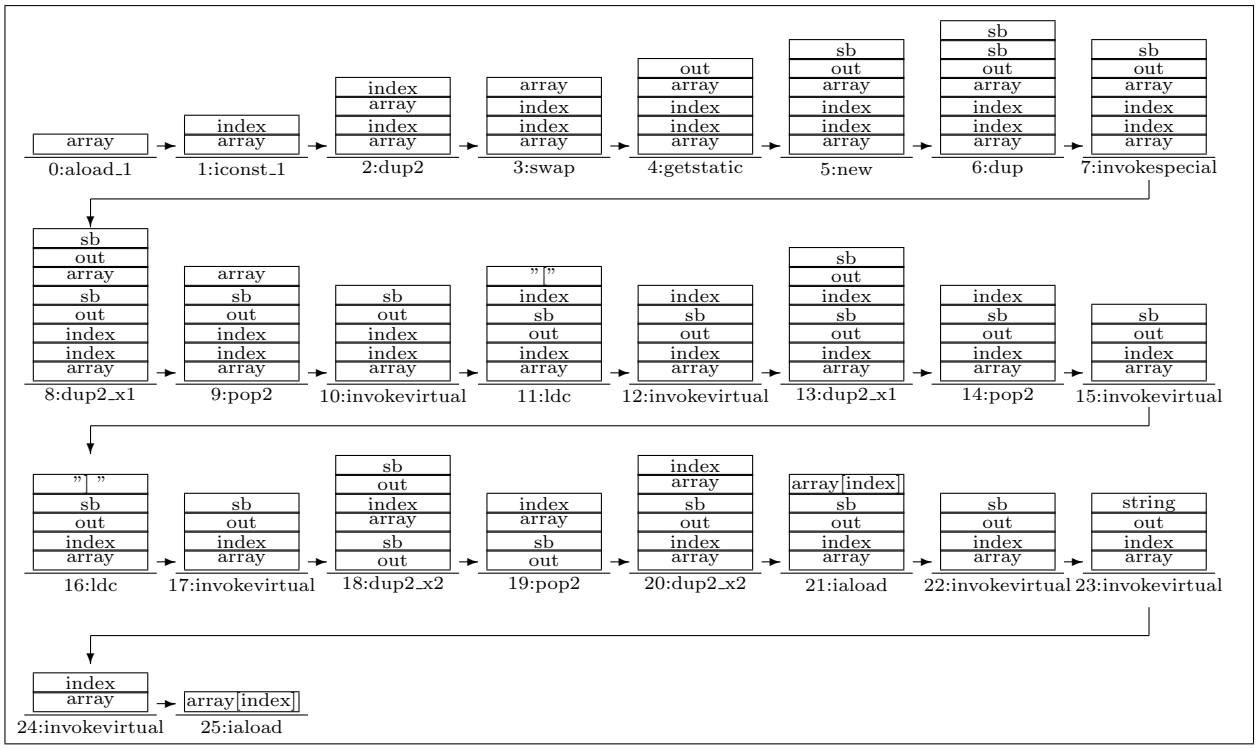


図 15　iaload に対するトレーサ挿入後のオペランドスタックの状態

算トレーサと変数への代入トレーサを挿入すればよい．

#### 3.3.4　条件分岐

プログラムでは if や else if などの条件分岐や for 文に代表される繰り返しがほぼ必ず使われる．ただ，ソースファイルでは条件分岐と繰り返しは明確に区別されているが，クラスファイルでは全てアドレスジャンプで実現されているため，区別されない．そして，このジャンプ先は相対アドレスで指定されているので，トレーサを挿入するとき，このアドレスの解決も同時に行われなければならない．

幸い，実装に用いたライブラリである BCEL はクラスを読み込んだとき，ジャンプ先をリンクとして扱うため，アドレスの付け替えはほとんど必要ない．しかし，ジャンプ先が return 文の場合のみ，ジャンプ先の変更が必要である．メソッドの終了のトレーサは，第 3.3.1 章で述べたように，return の直前に挿入される．しかし，ジャンプ命令で直接 return 命令に飛んでいた場合，メソッド終了トレーサが出力されない．そのため，ジャンプ先を調べて付け替える必要がある．

この作業は全てのトレーサが挿入し終わった後に行う．全てのメソッドの return 命令が targeter であるかどうかを調べる．targeter であるならば，ジャンプ先に指定されているため，ジャンプ先を変更する必要がある．他の全てのトレーサ挿入処理は，このとき，終了しているため，return 命令の直前に挿入しておいたトレーサコードが必ず存在する．そこで，return へジャンプしていた命令のジャンプ先を挿入したトレーサコードの先頭に付け替える．こうすることで，どのような場合でもメソッド終了トレーサが出力できるようになる．

## 3.4　実行結果

実行結果の例として，図 18 に示す Fibonacci に AddTracer を用いてトレーサを挿入した．そして，得られたクラスファイルから作成されたオブジェクトに対して，fibonacci メソッドを引数 4 で呼び出したときの出力が図 19 である．

```
dup2_x2
dup2_x2
pop2
dup2_x2
swap
getstatic java/lang/System#out
new java/lang/StringBuffer
dup
invokespecial java/lang/StringBuffer#<init>()V
dup2_x1
pop2
invokevirtual java/lang/StringBuffer#
    append(Ljava/lang/Object;)Ljava/lang/StringBuffer;
ldc "["
invokevirtual java/lang/StringBuffer#
    append(Ljava/lang/String;)Ljava/lang/StringBuffer;
dup2_x1
pop2
invokevirtual java/lang/StringBuffer#
    append(I)Ljava/lang/StringBuffer;
ldc "]: "
invokevirtual java/lang/StringBuffer#
    append(Ljava/lang/String;)Ljava/lang/StringBuffer;
dup2_x2
pop2
invokevirtual java/lang/StringBuffer#
    append(<basic type of array>)Ljava/lang/StringBuffer;
invokevirtual java/lang/StringBuffer#
    append(<type of array>)Ljava/lang/StringBuffer;
invokevirtual java/lang/StringBuffer#
    toString()Ljava/lang/String;
invokevirtual java/io/PrintStream#
    println(Ljava/lang/String;)V
dup2_x2
pop2
```

図 17　2 ワードの場合の配列への代入

```
0: public class Fibonacci{
1:     public int fibonacci(int n){
2:         if(n == 1 || n == 2){
3:             return 1;
4:         }
5:         return fibonacci(n - 1) + fibonacci(n - 2);
6:     }
7: }
```

図 18　サンプルプログラム (Fibonacci 数列)

# 4　ケーススタディ

ここでは，AddTracer の 2 つのアプリケーションを紹介する．一つはプログラムの理解性を測るための手法の前段階として，もう一つは事前知識の必要ないデバッグ方法に必要な情報を AddTracer を用いて抽出する．

もちろん，AddTracer の適用例はここに挙げたものだけではなく，理解性評価，デバッグ，テスト，解析，耐タンパ性評価など多岐に渡る．

## 4.1　仮想メンタルシミュレーション

仮想メンタルシミュレーションモデル (Virtual Mental Simuration Model; VMSM) を用いてプログラムの理解性を測る手法が提案されている [8, 3]．VMSM では FIFO キューと変数の代入/参照の系列 (Assignment and Reference Trace; AR-trace) を用いて，人の短期記憶の働きをシミュレートする．人間は，変数の値を忘れた際に，その変数が最後に更新された場所までプログラムをバックトラックする．そのバックトラック距離が，メンタルシミュレーションのコストに大きく影響することが VMSM メトリクスによって示された．

このメンタルシミュレーションを AddTracer を用いることで簡易化することができる．AddTracer は変数の参照，代入，演算などのあらゆる扱いを変数名と共に出力する．そのため，AddTracer を用いてトレーサコードを埋め込んだ Java クラスファイルを実行するだけで，メンタルシミュレーションを行うために充分な情報が得られることになる．

```
start Fibonacci#fibonacci at line 3
n    assignment(line 3): 4
n    reference(line 3): 4
n    reference(line 3): 4
this reference(line 6): Fibonacci@1e893df
n    reference(line 6): 4
-    operation(line 6): 3
start Fibonacci#fibonacci at line 3
n    assignment(line 3): 3
n    reference(line 3): 3
n    reference(line 3): 3
this reference(line 6): Fibonacci@1e893df
n    reference(line 6): 3
-    operation(line 6): 2
start Fibonacci#fibonacci at line 3
n    assignment(line 3): 3
n    reference(line 3): 3
n    reference(line 3): 3
end Fibonacci#fibonacci at line 4
this reference(line 6): Fibonacci@1e893df
n    reference(line 6): 3
-    operation(line 6): 1
start Fibonacci#fibonacci at line 3
n    assignment(line 3): 1
n    reference(line 3): 1
end Fibonacci#fibonacci at line 4
+    operation(line 6): 2
end Fibonacci#fibonacci at line 4
this reference(line 6): Fibonacci@1e893df
n    reference(line 6): 4
-    operation(line 6): 2
start Fibonacci#fibonacci at line 3
n    assignment(line 3): 2
n    reference(line 3): 2
n    reference(line 3): 2
end Fibonacci#fibonacci at line 4
+    operation(line 6): 3
end Fibonacci#fibonacci at line 6
```

図 19　図 18 の実行例

## 4.2　事前知識の必要ないデバッグ手法

プログラムのバグを発見し，それを除去するためには通常，対象プログラムとその仕様に関する深い理解が必

要である．特に，エラーと正常動作の境目となる入力値をテストすることは境界値テストと呼ばれ，非常に重要なテストの一つとなっている．この境界値テストを入力値に関する知識を必要とせず，プログラムコードのみから行うデバッグ手法が提案されている [4].

この手法では，スペクトルと呼ばれるプログラムの特徴を抽出し，そこからモデルを構築して，そこからバグの位置を特定する．Renieris らは以下の 2 つのスペクトルを提案している．

Coverage **スペクトル**　実行された命令のシーケンス
Permutation **スペクトル**　実行された命令とその回数

このスペクトルを抽出するにはプログラムの動的な解析が必要であり，抽出作業を自動化できるツールが不可欠である．そこで，AddTracer を用いることで，Java 言語で実装されたプログラムに対して，この手法を適用することが可能になる．

## 5　関連研究

松本らは Java 仮想マシンを拡張する方法で動的解析を可能にする DataExtracter を提案している [6]．このツールは AddTracer と同じく，プログラムの実行途中のデータを捕捉することを可能にする．しかし，AddTracer とはアプローチが異なり，Java Platform Debugger Architecture (JPDA) を拡張することで実現されている．また，DataExtraceter は HotSpot に対応できないと述べられており，この点が DataExtracter の弱点となっている．

対する AddTracer は Java のバイトコードを変換する単なるコンバータであり，Java の実行環境に如何なる変更も加えることがないため，よりポータビリティが高いと言える．また，AddTracer はクラスファイルのフォーマットのみに依存するため，HotSpot であろうが，JIT であろうが，どのような Java 仮想マシンであっても動作する．

## 6　まとめ

本稿では Java クラスファイルのバイトコードを編集することで，プログラムの状態を出力するトレーサを挿入する方法を示した．この実現により，プログラムの耐タンパ性の評価を行うことができる．

耐タンパ性評価以外の用途として，メンタルシミュレーションやデバッグを行うための足掛かりとしても使えることをケーススタディで示した．

今後の課題として，より細かな命令に対するトレーサの埋め込みと，出力されたトレーサを如何に扱うかの方法論に関する議論を行う必要があるだろう．

## 参考文献


[1] Jakarta BCEL. http://jakarta.apache.org/bcel/.

[2] Tim Lindholm and Frank Yellin. *The Java$^{TM}$ Virtual Machine Specification Second Edition*. Addison-Wesley Pub Co, Apr 1999.

[3] Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken ichi Matsumoto, Yuichiro Kanzaki, and Hirotsugu Satoh. Queue-based cost evaluation for mental simulation process in program comprehension. In *Proc. 9th International Software Metrics Symposium (METRICS2003)*, pages 351–360, Sep 2003.

[4] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proc. 18th IEEE International Conference on Automated Software Engineering, 2003*, pages 30–39, Oct 2003.

[5] Haruaki Tamada. AddTracer: Injecting tracers into java class files for dynamic analysis, Dec 2004. http://se.aist-nara.ac.jp/addtracer/.

[6] 松本 勉, 赤井 健一郎, 中川 豪一, 大内 功, 竹脇 和也, and 村瀬 一郎. Java 対応ランダムデータ補足ソフトウェア. **情報処理学会論文誌**, 44(8):1947–1954, Aug 2003.

[7] 鷲見 豊, editor. *Java* **バーチャルマシン**. オライリー・ジャパン, Jul 1998.

[8] 石黒 誉久, 井垣 宏, 中村 匡秀, 門田 暁人, and 松本 健一. 変数更新の回数と分散に基づくプログラムのメンタルシミュレーションコスト評価. In **電子情報通信学会 技術研究報告 ソフトウェアサイエンス研究会**, volume SS 2004-32, pages 37–42, Nov 2004.