チュートリアル

プログラムの動的解析

石尾 隆

動的解析とは，プログラムの実行時情報を収集し，その実際の振舞いや性能を解析するための技術である．本稿では，ソースコードに対する解析と比べた場合の動的解析の特徴を述べ，コードカバレッジの計算や統計的デバッギングなどの動的解析手法を解説する．また，読者が解析手法を試すことができるように，Java プログラムの解析に役立つツールについても紹介する．

Dynamic analysis is a category of program analysis techniques that analyze execution traces of a program to understand the behavior and the performance of the program. This paper explains several applications of dynamic analysis such as code coverage analysis and statistical debugging and their effectiveness. The paper also presents several available tools to analyze Java programs.

1 はじめに

プログラム解析（Program Analysis）は，人間が作成したプログラムの記述や構造，振舞いを解析し，ソフトウェア開発に有益な情報を抽出するための技術である．プログラム解析技術のうち，解析対象プログラムのソースコードや設計文書などのプロダクトを解析する技術を静的解析（Static Analysis），プログラムの実行を解析する技術を動的解析（Dynamic Analysis）と呼ぶ．動的解析手法の多くは静的解析を併用するため，境界は明確ではないが，最終結果を得るために解析対象プログラムを 1 度でも実行する必要がある技術は動的解析に属していると考えるとよい．たとえば，本稿で紹介するコードカバレッジの計算は，プログラムに対するテスト実行から，テストの実施状況の指標を求める動的解析手法である．

動的解析の主な利用目的は，テストおよびデバッグである．テストとは，対象プログラムの動作と開発者が期待する動作とが一致しているかを調査する行為であり，デバッグとは，期待した動作と実際の動作が異なる原因を調べる行為である．動的解析の技術のいくつかは，これらの疑問に答えるための情報を効果的に収集する手段を提供する．その他の用途としては，現在稼働しているプログラムの動作の可視化や，実行時性能の分析などが挙げられる．

本稿では，動的解析の例として，コードカバレッジの計算，統計的デバッギング，動的プログラムスライシングを紹介する．それぞれの手法は，現在も様々な技術が開発されているが，本稿では，解析の仕組みの理解しやすさを優先し，最新の動向ではなく，古典的な手法と具体例に基づいて解説を行う．

本稿で登場する例やツールは，特に明記していない限り，すべて Java を対象とした解析となっている．Java は，プラットフォーム非依存で振舞いが定義されており，処理系による差異の少なさ，実装のしやすさから，最も活発に動的解析が研究されている言語である．ツールも数多く提供されていることから，動的解析を学習するのにも適している．

本稿の構成は次の通りである．まず 2 章で用語を確認してから，3 章で動的解析の特徴を，4 章で代表

Dynamic Program Analysis

Takashi Ishio, 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.16, No.5 (1999), pp.78–83.
[解説論文] 1999 年 8 月 3 日受付.

的な手法を紹介する．5 章では動的解析の実現に役立つツールを紹介し，6 章でまとめを述べる．

2 用語

本稿では，プログラムを，ある特定の環境で，特定の入力データの列を与えて 1 回実行するとき，その実行環境および入力データの列を指して**テストケース（test case）**と呼ぶ．動的解析の適用目的はテストに限らないが，プログラムの特定の機能を実行するために開発者が用意する入力データという意味で，多くの論文でも使用されている表現である．

あるテストケースの実行開始から終了までに実行された命令の列と，そのときのプログラムの状態の系列のうち，解析に必要な情報を記録したものを，そのテストケースに関する**実行トレース（exexcution trace）**と呼ぶ．実行トレースに含まれる情報は解析手法によって異なり，たとえば，4.1 節で紹介するコードカバレッジの計算では，実行された行番号の系列を実行トレースとする．また，メソッドの呼び出し関係を調べる手法では呼び出されたメソッドの名前の系列が，実行時のデータの流れを追跡する手法では，各命令がアクセスした変数やオブジェクトの ID，配列の添え字といった情報の系列が，それぞれ実行トレースとなる．

3 動的解析の特徴

動的解析は，プログラムの実行トレースとして記録された，実行中に起きた現象を解析する技術である．ソースコードを対象とする静的解析技術との違いを表 1 に示す．静的解析と動的解析の最大の違いは，静的解析が「プログラムが起こしうる動作」を解析する手法であるのに対し，動的解析が「プログラムが実際に起こした動作」を解析することである．静的解析では，プログラムが起こしうる動作を見逃さないように，プログラムの動作を近似的に計算するため，現実には起こらない動作まで計算結果に含めてしまうことがある．一方，動的解析は，プログラムの実行を解析しているので，プログラムの動作を正確に反映した結果が得られる．ただし，動的解析の結果は，毎回同一であるとは限らない．テストケースが同一であっても，それを実行した日時の情報や，スレッドの実行スケジュールによってプログラムの動作が変化し，異なる結果を出力する可能性がある．

```
 1: public class Number {
 2:   public static void main(String[] args) {
 3:     Number v = new Number();
 4:     for (int i=0; i<args.length; ++i) {
 5:       v = new Succ(v);
 6:     }
 7:     System.out.println(v.value());
 8:   }
 9:   public int value() {
10:     return 0;
11:   }
12: }
13: public class Succ extends Number {
14:   private Number base;
15:   public Succ(Number base) {
16:     this.base = base;
17:   }
18:   public int value() {
19:     return base.value() + 1;
20:   }
21: }
```

図 1 動的束縛を使ったプログラムの例

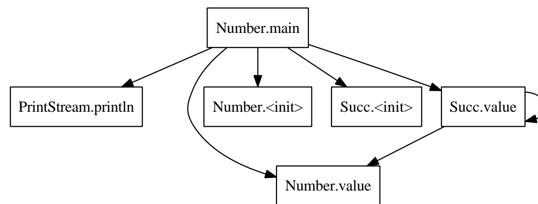

図 2 図 1 のプログラムの静的呼び出し関係グラフ．<init> はコンストラクタを表す．

静的解析と動的解析の違いを示す例として，Java におけるメソッド呼び出しの解析を挙げる．静的解析を用いた呼び出し関係の抽出は，1 つのメソッド呼び出し文に対応して実行されうるすべてのメソッドを列挙する．これに対して，動的解析では，ある特定のテストケースでの呼び出し関係を抽出する．図 1 に示すプログラムは，3 行目で変数 `v` に `Number` オブジェクトを代入するが，コマンドライン引数が存在する場合は変数 `v` の中身を 5 行目で作成した `Succ` オブジェクトに入れ替える．7 行目での `v.value()` メソッドの呼び出しは，動的束縛の対象である．すなわ

表 1　静的解析と動的解析が持つ主な性質

比較項目	静的解析	動的解析
解析に必要なもの	コンパイル可能なソースコード	実行可能なプログラム，テストケース
解析対象となる動作	プログラムが起こしうる動作	プログラムが起こした動作
解析結果の再現性	解析対象が変更されない限り有効	解析結果が毎回同一であるとは限らない
解析の基盤ツール	コンパイラ	デバッガ
利用可能な情報の例	ソースコードの位置情報，制御構文，データフローグラフ，コメントの内容	命令の実行順序，実行回数，実行時間，変数の実際の値

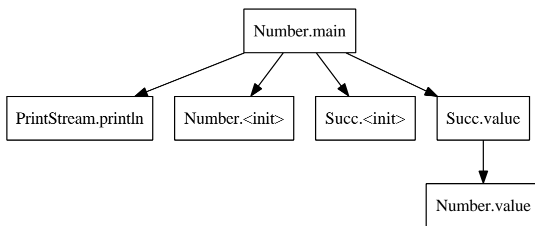

図 3　図 1 のプログラムに引数を 1 つ与えて実行したときの動的呼び出し関係グラフ

ち，この時点で変数 `v` に格納されているのが `Number` オブジェクトであれば `Number` クラスに定義された `value()` メソッドを，`Succ` オブジェクトであった場合は `Succ` クラスに定義された `value()` メソッドを実行する．そのため，静的な呼び出し関係の解析 [3] [28] では，図 2 に示すような呼び出し関係が得られる．これに対して，1 つのテストケースとして，コマンドライン引数を 1 つだけ与えてプログラムを実行した場合の動的な呼び出し関係の解析を行うと，5 行目が 1 度だけ実行され，`Succ` オブジェクトが `v` に代入されるため，図 3 に示すような呼び出し関係が得られる．この呼び出し関係グラフは，実行されなかった処理，たとえば引数が与えられなかった場合の `Number.main` メソッドから `Number.value` メソッドへの呼び出しや，引数が 2 個以上あった場合に生じる `Succ.value` メソッドから自分自身への呼び出しに関する情報を含まない．つまり，一部の呼び出しは見逃していることになるが，引数が 1 つの場合における正確な振舞いを示したグラフとなり，この実行に関してデバッグ等の調査を行いたい開発者にとっては，静的解析よりも有益な情報となる．

一般に，プログラムが取りうるあらゆる振舞いを網羅するようにプログラムを実行することができれば，その動的解析の結果は，静的解析の結果に近づいていくことになる．静的解析では，プログラムの実行経路を列挙する際に，実際には発生しえない実行経路を多数抽出する場合がある [19] ことから，最終的な結果が完全に一致するとは限らない．多数のテストケースを用いた動的解析のほうが，プログラムの実際の動作に近い結果を得られる場合もある．ただし，あらゆるテストケースを網羅的に実行することは不可能であり，見逃しが生じる可能性を考慮する必要がある．スレッド間のデッドロックやメモリリークのように，見逃しが重大な障害を引き起こすと考えられる欠陥がプログラムに含まれているかどうかを調査する場合は，プログラムの動作を網羅するために静的解析が用いられることが多い．

動的解析は，実際の動作を解析するという特徴に基づいて，様々な目的に利用されている技術である．その種類は多岐に渡っており，本稿ですべてを紹介することはできないが，代表的な手法を表 2 に挙げる．実行経路の分析は，動的解析の基本的な手法であり，観測した（あるいは観測中の）プログラムの動作を，開発者に有用な形で提示するための技術である．欠陥の分析はデバッグを目的とした手法で，欠陥の原因を特定するための技術や，欠陥が発生した状況を再現し分析するための技術がこれに含まれる．振舞い仕様のマイニングは，変数の値域やメソッドの実行順序など，ソースコードには書かれていない暗黙のルールを探索するための技術であり，主にプログラム理解に使われている．

動的解析手法には様々なものがあるが，プログラムの実行が完了してから実行トレースを解析する手法をオフライン解析（offline analysis），実行を継続し

表 2 代表的な動的解析手法

カテゴリ	手法の名称	手法の概要
実行経路の分析	コードカバレッジの計算	実行されたプログラム文の割合から，テストの進捗状況を示す数値を計算する．[4] [5]
	振舞いの可視化	プログラムの実行内容をシーケンス図等の形式で可視化する．[6] [21] [23] [29]
	フェイズ検出	プログラムが，ある処理の実行から別の処理に移ったタイミングを検出する．[24] [32]
欠陥の分析	統計的デバッギング	成功した実行と，失敗した実行の差から，欠陥の原因となっているコードを自動的に特定する．[13] [20] [35]
	動的スライシング	開発者が指定した変数の値に影響を与えたプログラム文を特定する．[1] [30]
	Capture & Replay	あるオブジェクトの入出力をすべて記録し，オブジェクトの状態を再現する．[14]
振舞い仕様のマイニング	プロトコルマイニング	単一あるいは少数のオブジェクトに対するメソッド呼び出し順序のルールを発見する． [11] [18]
	動的不変条件の検出	変数の実際の値から，各変数の値域や変数間の関係式などを不変条件として抽出する．[10]
	Live Sequence Chart の抽出	オブジェクト間のメソッド呼び出しパターンを解析し，パターンの出現順序を可視化する．[17]

た状態で実行トレースを解析する手法をオンライン解析（online analysis）と呼ぶ．オンライン解析は，その時点までに得られた部分的な実行トレース情報しか使用できないかわり，解析が終わった実行トレース情報を蓄積する必要がなく，また，解析結果をプログラムの実行制御などに反映することが可能であるという特徴を持つ[8]．表 2 に示した手法の中では，フェイズ検出手法[24]，プロトコルマイニング[11] がオンライン解析に属する．最近の論文では，どちらに属しているかを明記していることが多いので，実行を停止できないソフトウェアに対する解析技法に興味がある場合は，オンライン解析と記述された論文を調べるとよい．

4 動的解析手法の紹介

本章では，動的解析手法として，コードカバレッジの計算，統計的デバッギング，動的プログラムスライシングを紹介する．これらの手法は，いずれもアイディアは簡明であるが，現在の様々な研究の基盤となっている古典的な手法である．

各手法の具体的な手順を示すために，図 4 に示す例題プログラムを用いる．このプログラムは，うるう年の判定を実装しており，コマンドライン引数に 指定された数値 1 つを変数 year に代入し，その数値が

```
 1: public class LeapYear {
 2:   public static void main(String[] args) {
 3:     int year = Integer.valueOf(args[0]);
 4:     String answer;
 5:     if (year % 4 == 0) {
 6:       if (year % 100 == 0) {
 7:         if (year % 4*100 == 0) {
 8:           answer = "yes";
 9:         } else {
10:           answer = "no";
11:         }
12:       } else {
13:         answer = "yes";
14:       }
15:     } else {
16:       answer = "no";
17:     }
18:     System.out.println(answer);
19:   }
20: }
```

図 4 うるう年の判定を行うプログラム（誤りを含む）

表す年がうるう年なら yes，そうでなければ no を出力する．うるう年の条件は，4 の倍数でありかつ 100 の倍数ではないこと，あるいは 400 の倍数であることである．なお，この例題プログラムには，意図的に誤りを 1 つ混入させている．

4.1 行カバレッジの計算

最初に紹介する動的解析は，ソフトウェアテストの基礎となっている動的解析手法，行カバレッジ（Line Coverage）の計算である．

4.1.1 解析の目的

コードカバレッジとは，十分なソフトウェアテストを実施したかを評価するための網羅性の指標である．その 1 つ，行カバレッジ（Line Coverage）は，テストケース集合によって実行されたプログラムの行番号の集合が，実行可能な命令行に対して占める割合を求めたものである．この割合が 100% になったということは，すべての行が少なくとも 1 回はテスト実行されたことになる [4]．そのため，行カバレッジの計測は，テストの実施状況を知る重要な解析手法となっている．本稿では，テスト技法そのものを扱うわけではないので，行カバレッジを 100%に近づけるようにテストケースを作成する方法に関しては取り扱わない．

4.1.2 解析の方法

テスト対象プログラムの実行可能な命令行の集合を L とすると，ある 1 つのテストケース t に対する行カバレッジ C_t は，t に関するプログラムの実行トレースに含まれる行の集合 L_t によって，次のように求められる．

$$C_t = \frac{|L_t|}{|L|}$$

図 4 に示すプログラムの場合，変数宣言や記号（“}”）だけの行を取り除くことで，実行可能な命令を持つ行の集合 $L = \{3, 5, 6, 7, 8, 10, 13, 16, 18\}$ が得られる．ここで，入力値 year として 2011 を与えてプログラムを実行すると，$L_{year=2011} = \{3, 5, 16, 18\}$ となることから，このテストの行カバレッジの値は次のようになる．

$$C_{year=2011} = \frac{|L_{year=2011}|}{|L|} = \frac{4}{9} = 0.444\cdots$$

テストケース集合 T が与えられたときは，個々のテストケースで実行された行の和集合を使って行カバレッジ C_T の値を求める．

$$C_T = \frac{|\bigcup_{t \in T} L_t|}{|L|}$$

行カバレッジの値を計算するには，プログラムに含まれた命令行の集合と，実際に実行された命令行の集合が必要である．前者は静的解析によって，後者はプログラム実行の観測によって求める．

4.1.3 Cobertura を使ったカバレッジの計算

行カバレッジの計算は，ソフトウェアテストの基礎となっていることもあり，様々なソフトウェアに組み込まれている．本稿では，無償で使えるソフトウェアである Cobertura[†1] を用いて，例題プログラムを解析する．Cobertura を使用したカバレッジ計算の手順は，次のようになる．

1. 解析対象プログラムに実行トレースを記録するための命令を埋め込む．
2. 1 つ以上のテストケースを実行し，実行トレースを記録する．
3. 実行トレースから，カバレッジ情報のレポートを生成する．

行カバレッジの計算には，プログラムに含まれた命令行の集合と，実行された行の集合が必要である．そのため，まず，解析対象プログラムを分析し，実行可能なすべての命令行を列挙し，実行トレースを記録するための命令を埋め込む．Microsoft Windows 環境では，Cobertura に付属しているバッチファイルを以下のように実行する．

```
cobertura-instrument.bat
    --destination instrumented bin
```

コマンドに与えている引数--destination は，出力先の指定である．bin ディレクトリ以下に配置されているプログラムを入力として，解析用命令を追加したプログラムを instrumented という名前のディレクトリに書き出す．このとき，処理を実行したディレクトリには，プログラムの情報を格納した cobertura.ser というファイルが生成される．

Cobertura は Java バイトコード変換を採用しており，コンパイル後のバイトコードに埋め込まれている命令と行番号の対応情報を使って，命令行の集合を取得し，各命令の直前に，実行トレースを記録するための命令を埋め込む．図 4 に示すコードにこの変換

†1 http://cobertura.sourceforge.net/

を適用した後，Java Decompiler †2 を使ってソースコードに変換し，行カバレッジに関するコードを抽出した結果を図5に示す．行末に“*”を付与した行が，Cobertura が挿入した命令である．このソースコードを見ると，各行の命令の直前に，クラス名と行番号を引数としたメソッド呼び出しが挿入され，どの行を実行したかを記録するようになっていることが分かる．なお，Cobertura はブランチカバレッジも計算するため，実際の生成コードには，さらに条件分岐に関するロギング処理も加わる．

実行トレースの記録処理を埋め込んだプログラムを用いて，通常のプログラム実行と同様にテストケースの実行を行う．この実行では，たとえば以下のように，Cobertura のライブラリファイルと変換後のクラスファイルを指定してプログラムを実行する．

```
java -cp "cobertura.jar;instrumented"
    LeapYear 2011
```

プログラムの実行トレースは，バイトコード変換の際に生成された `cobertura.ser` というファイルに保存される．以下のコマンドで，このファイルとソースコードを読み込み，引数`--destination` で指定されたディレクトリに HTML 形式のレポートを出力する．

```
cobertura-report.bat
    --destination report src
```

例題プログラムに引数 2011 を与えて実行したときのカバレッジの出力例を図6に示す．ソースコードの左端に行番号と，その行の実行回数が表示されており，まだ実行されていない行，条件分岐の `true` と `false` のどちらか一方がまだ実行されていない行が赤色で強調表示されている．この図において，1行目のクラス宣言と，19行目のメソッド終了の中括弧が実行可能命令として表示されているが，これは Cobertura が Java バイトコードを解析対象としているために生じている現象である．1行目は `LeapYear` クラスのソースコードには宣言されていない暗黙のコンストラクタに対応しており，19行目は戻り値のないメソッドで省略されている（がバイトコード上では命令として出現している）`return` 文に対応している．このよ

†2 http://java.decompiler.free.fr/

```
public static void main(String[] args)
{
  TouchCollector.touch("LeapYear", 3);        *
  int year = Integer.valueOf(args[0]);
  String answer;
  TouchCollector.touch("LeapYear", 5);        *
  if (year % 4 == 0) {
    TouchCollector.touch("LeapYear", 6);      *
    String answer;
    if (year % 100 == 0) {
      TouchCollector.touch("LeapYear", 7);    *
      String answer;
      if (year % 4 * 100 == 0) {
        TouchCollector.touch("LeapYear", 8);  *
        answer = "yes";
      } else {
        TouchCollector.touch("LeapYear", 10); *
        answer = "no";
      }
    } else {
      TouchCollector.touch("LeapYear", 13);   *
      answer = "yes";
    }
  } else {
    TouchCollector.touch("LeapYear", 16);     *
    answer = "no";
  }
  TouchCollector.touch("LeapYear", 18);       *
  System.out.println(answer);
}
```

図5 Cobertura によってロギング命令が挿入されたコード．挿入された命令を行末の“*”で示している．

うなバイトコード上の表現の影響で，4.1.2 項で示したカバレッジの値は9行中の4行を実行していたが，Cobertura の出力は11行中の5行となっている．

Cobertura は出力ファイル `cobertura.ser` に実行トレースのデータを蓄積していく．複数のテストケースがある場合は，それらを順番に実行していけば，カバレッジの値が増加していく様子を確認することができる．蓄積されたカバレッジ情報を破棄する場合は，この出力ファイルを削除してから `cobertura-instrument.bat` を再実行し，プログラムを一度も実行していない状態のファイルを作り直せばよい．

4.1.4 手法の現状

行カバレッジの計算は非常に重要な意義を持ち，多くのプログラミング言語で，様々なツールが公開され

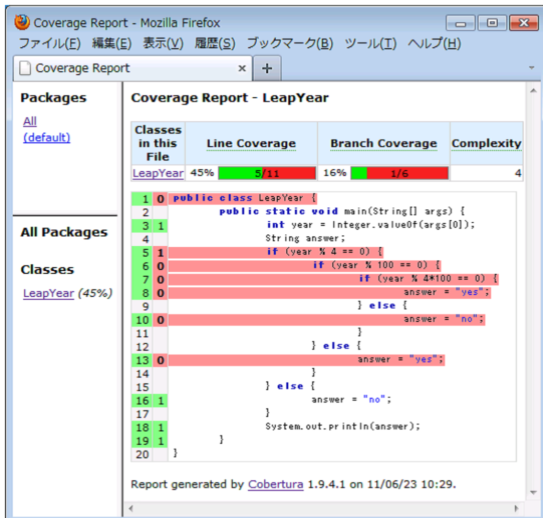

図 6　Cobertura の出力した HTML ファイルの例．例題プログラムの入力 year に 2011 を指定した場合のカバレッジ情報を表示している．

ている．既に普及した技術の 1 つではあるが，行カバレッジの計算を適用する際には，2 つの注意点が存在する．

まず，行カバレッジが 100%になっても，プログラムの実行条件を完全に網羅しているわけではない．行カバレッジはプログラムの各行を 1 回だけ実行すればよいので，たとえば “`if (x) y=readData();`” というように `else` 節を伴わない条件分岐が存在すると，条件が真の場合のみ実行すれば，行カバレッジは達成されることになる．このような行カバレッジの弱点を補う指標として，条件分岐で使われた真偽値の網羅性を用いるブランチカバレッジ（Branch Coverage）や，実行パスを区別するパスカバレッジ（Path Coverage）などが存在する [5]．ただし，これらの指標も完全ではなく，たとえばブランチカバレッジの場合は，条件分岐以外，たとえば動的束縛や例外処理によって生じる実行の分岐は考慮されていないことが多い．

カバレッジに関するもう 1 つの問題は，ソースコードと実行バイナリの差異である．本稿で示した例のように，クラスにコンストラクタが宣言されていない場合，Java コンパイラは引数を持たないコンストラクタを生成してしまう．そして，オブジェクトを生成しないクラスに関しては，コンストラクタが実行されていないために，行カバレッジが 100%に到達しないという結果になる．また，過剰なエラーチェックなど，意味のない条件分岐が記述されると，ブランチカバレッジも 100%に到達しないという状況が起きうる．100%に近づいたカバレッジが，既に十分な値であるかどうかを判断するには，コンパイラの挙動や対象プログラムの内容に関する知識が必要となる．

カバレッジ計算に関しては，実行トレースの計測コストを削減するための研究が実施されている．その 1 つは，データフローに関するカバレッジを計算する際，実行トレースとして記録しなければならないデータ量を静的解析によって減らす手法である [26]．また，Java を対象とした手法ではないが，ある一定間隔に時間を区切り，その区間内では実行順序を無視して，実行していたコードの位置を高速に記録する手法が提案されている [9]．

4.2　統計的デバッギング手法

テストを実行していくと，プログラムの誤りに行き当たることがある．図 4 のプログラムは，2011 に対しては正しく `no` を，2012 を入力した場合は正しく `yes` を出力する．しかし，2100 を入力した場合は，2100 年はうるう年ではないにも関わらず，`yes` を出力してしまう．普通の開発環境であれば，デバッガを起動し，この出力が得られた理由を分析することになるが，本節では，行カバレッジの情報を用いた統計的デバッギング手法の適用例を示す．

4.2.1　解析の目的

統計的デバッギング（Statistical Debugging）[36] は，プログラムの欠陥の原因を特定する Fault Localization と呼ばれる技術の 1 つで，プログラムの実行に成功した場合と失敗した場合の実行時情報が与えられたとき，それらの振舞いの差分から，欠陥の原因となっているプログラム文（あるいはデータフロー等の問題となる振舞い）を特定する手法である．開発者は，プログラムに対して適切なテストを実行しさえすれば，欠陥の原因として疑わしいプログラム文の集合を特定することができ，それらのプログラム文を優先

的に調査することが可能となる．

4.2.2 解析の方法

統計的デバッギングの基本的な考え方は，Jones ら [13] が提案している Tarantura というツールに実装された，次のようなものである．

- ある文を実行したテストケースの多くが，正しい結果を出力していれば，その文と欠陥との関係は弱い．
- ある文を実行したテストケースの多くが，誤った結果を出力していれば，その文と欠陥との関係は強い．

Tarantura は，この考えに基づいて，プログラムに含まれる各文の安全度を計算する．この計算を行うために，まず，開発者は，プログラムに対するテストケース集合 T を用意し，各テストケース $t \in T$ を実行する．このとき，各テストケースについて個別に行カバレッジを計測し，また，出力結果が予測通りであったものを成功テストケース P (passed)，そうでなかったものを失敗テストケース F (failed) として分類する．P，F はそれぞれ少なくとも 1 つ存在しなければならない．

テスト実行が終わったら，プログラムの各文 s について，P に属する成功テストケースのうち，プログラム文 s を実行したものの割合を計算し，$\%passed(s)$ とする．また，F に属する失敗テストケースのうち，プログラム文 s を実行したものの割合を計算し，$\%failed(s)$ とする．そして，プログラムの各文 s の安全度 $Safety(s)$ を，以下の式で求める．

$$Safety(s) = \frac{\%passed(s)}{\%passed(s) + \%failed(s)}$$

Tarantura は，2011 年 9 月現在，残念ながら公開されていないが，テストケースごとの行カバレッジの情報を取得することができれば，テストケースごとの成功か失敗かの判定を人間が指定することで，$Safety(s)$ の値を計算することができる．図 4 のプログラムに対して 2011, 2012, 2100 を入力し，実行結果から手作業で Safety の値を計算した結果を表 3 に示す．ここでは，7 行目および 8 行目の $Safety$ の値が 0 となっており，欠陥の可能性が最も高いことを示している．実際に，この問題の原因は 7 行目の条件分岐にある．剰余演算子 “`%`” と乗算演算子 “`*`” の優先順位は同じであるから [12]，“`year % 4*100`” という式は “`(year % 4)*100`” と解釈され，7 行目の条件式は常に真となって 8 行目の処理が実行されている．

統計的デバッギングの利点は，多数のテストケースが存在しているとき，疑わしいプログラム文を自動的に抽出できることである．ただし，Tarantura の使う計算式はテストケースごとの行カバレッジの差に基づいているため，成功テストケースと失敗テストケースのカバレッジが一致するとき，欠陥の原因を特定できない．たとえば，入力として 2000 を与えた場合のカバレッジは 2100 の場合と完全に一致するが，出力は `yes` で正しい．そのため，2011, 2012, 2100, 2000 の 4 つのテストケースから計算を行うと，7 行目，8 行目に対する $\%passed$ の値が 0.33 となり，これらの行に対する $Safety$ の値が上昇する（プログラム中で最も低い値であることに変わりはない）．

4.2.3 手法の現状

統計的デバッギングに関しては，様々な拡張が存在しており，条件分岐によるコードブロック間の制御の移動に着目したモデル [35] や，データアクセス順序による並列プログラムのデバッグ [20] などがある．統計的デバッギングは，故障の発生に再現性がない場合であっても，故障が発生した場合の実行トレースが得られれば，故障が発生しなかった場合の実行トレースと比較することで，その原因に近づける可能性がある．そのため，現在も活発に研究が進められている．

統計的デバッギングの弱点は，4.2.2 項でも述べた通り，問題のある命令を実行しても，その影響が検出できなかったテストケースは成功として扱うため，原因の特定を阻害してしまうことがある．この問題は Coincidental Correctness として知られており，実行トレースとして記録する情報を増やすことで対策する手法が提案されている [31].

4.3 動的プログラムスライシング

4.2 節で紹介した統計的デバッギングは，プログラムの中から欠陥の原因である可能性がある文を提示

表 3 Tarantura の $Safety$ 値の計算例

	Line Coverage					
Source Code	2011	2012	2100	$\%passed$	$\%failed$	$Safety$
3: int year = Integer.valueOf(args[0]);	x	x	x	1.0	1.0	0.5
4: String answer;						
5: if (year % 4 == 0) {	x	x	x	1.0	1.0	0.5
6: if (year % 100 == 0) {		x	x	0.5	1.0	0.33
7: if (year % 4*100 == 0) {			x	0	1.0	0
8: answer = "yes";			x	0	1.0	0
9: } else {						
10: answer = "no";						
11: }						
12: } else {						
13: answer = "yes";		x		0.5	0	1.0
14: }						
15: } else {						
16: answer = "no";	x			0.5	0	1.0
17: }						
18: System.out.println(answer);	x	x	x	1.0	1.0	0.5
19: }						
期待される出力:	no	yes	no			
実際の出力:	no	yes	yes			
実行の成否:	P	P	F			

する手法であるが，実際にどの文が原因で，なぜ問題が起きるかを調べるのは開発者の仕事である．プログラムの実行から，欠陥の原因を分析する技術の 1 つとして，動的プログラムスライシングを紹介する．

4.3.1 解析の目的

動的プログラムスライシング（Dynamic Program Slicing）[1] は，ある特定のテストケースに対して，ある実行時点での変数の値に影響を与えた命令列を特定するための手法である．影響を与えた命令とは，その変数の値を計算した命令と，命令を実行するかどうかを判定した条件分岐命令である．変数の値は，通常，他の変数の値から計算されるので，時系列をさかのぼる形で推移的に影響を探索していき，なぜその値が計算されたのかを示すことが，スライシングの目的である．

動的プログラムスライシングもまた，デバッグ支援手法であるが，統計的デバッギングとは異なり，単体のテストケースでの解析を行う．

4.3.2 解析の方法

動的プログラムスライシングは，実行された命令間でのデータ依存関係，制御依存関係を計算する．まず，データ依存関係 DU (Definition-Use) と制御依存関係 TC (Test-Control) を次のように定義する [27].

- ある命令の実行 p が代入した変数 v の値が，それ以降の命令の実行 q までに上書きされることなく，q で使用されたとき，関係 $DU(p,q)$ が成り立つものとする．
- ソースコード上で，実行制御文 t の条件節の結果が命令 s を実行するかしないかを決定するとき，命令 s のある実行時点を q とすると，q より前で q に最も近い t の実行時点 p について，関係 $TC(p,q)$ が成り立つものとする．

関係 $DC(p,q)$ および $TC(p,q)$ は，それぞれ p から q への有向辺と考えると，実行時点を頂点とした有向グラフと考えることができる．このグラフを動的依存グラフと呼ぶ．例として，図 4 のプログラムに 2100 を入力した場合の実行系列に対する動的依存グラフを図 7 に示す．図 7 において，実線の辺は，ラベルに書かれた変数の値が到達したことによる DU 関係を表現しており，control というラベルの付いた点線の辺は TC 関係を表現している．

動的プログラムスライシングは，実行された命令列を，時系列と逆方向にさかのぼる形で，問題の原因へと近づいていくアプローチである．開発者は，正し

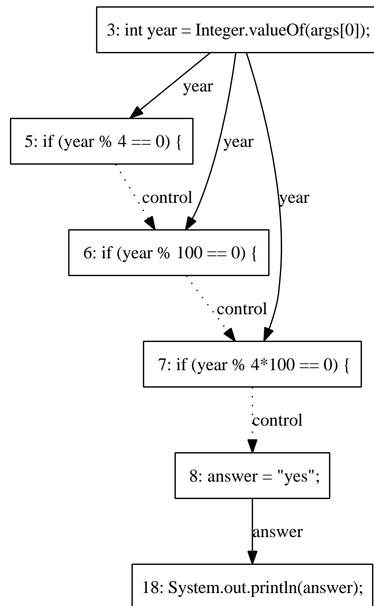

図 7 y=2100 の実行に対する動的依存グラフ

くない出力結果が得られた 18 行目の実行を基点に，`answer` が 8 行目から来たこと，その実行が 7 行目の `if` 文によって定められたことをグラフ探索によって確認することができる．通常のデバッガが，ブレークポイントとステップ実行を用いて問題を起こした命令を特定する必要があるのに対し，問題が生じた時点からさかのぼる探索となるため，原因の分析を容易に行うことができる．

動的プログラムスライシングでは，データがどこで計算され，どこで使われたかを追跡する必要があるため，実行トレースとして，オブジェクトのフィールドや配列の要素へのアクセスを記録しておく必要がある．そのため，実装上は，計算コストが非常に高いのが弱点である．オブジェクト ID などを実行トレースとして記録するためのコストについては，櫻井ら [25] が Traceglasses というデバッグ支援ツールの提案の中で実験しており，プログラムの実行時間がおよそ 4 倍から 14 倍，最大で 93 倍になったと報告している．櫻井らの手法自体は動的プログラムスライシングとは異なるが，この結果を踏まえると，実行トレースを記録する処理の実行時間が数秒程度と短い場合には，動的プログラムスライシングは適用可能であるが，長時間の実行が必要な処理に関しては，実行のオーバーヘッドが通常のツールのみで行うデバッグ作業時間よりも長くなり，適用が困難であると考えられる．

4.3.3 手法の現状

動的プログラムスライシングのように，プログラムの実行を過去にさかのぼるアプローチは，デバッグにおいて有望な技術として，活発に研究が行われている．

動的プログラムスライシングの拡張の 1 つとして，変数の値まで実行トレースに記録しておき,「なぜ変数 `answer` には “yes” が代入されたか」といった対話的な質問方式で，動的依存グラフの辺を辿っていく作業を支援する手法が提案されている [15]．また，スライシングのように過去にさかのぼるアプローチを突き詰め，実行トレースとして各命令を実行したときのメモリの完全な情報を保存し，プログラムの任意の時点の変数の状態を分析する Omniscent Debugging [16] も提案されている．

これらの手法の実用化に向けての最大の障壁は，記録すべきデータ量が非常に多いことである．動的プログラムスライシングに対しては，観測する必要があるデータを事前に計算しておく手法 [34] が提案されている．また，Omniscent Debugging に対しては，実行トレースを完全に記録するかわりに,「過去の状態に戻す命令」を生成しておき，過去の状態が必要なときにそれらの命令を実行する手法 [2] が提案されている．

5 動的解析手法の構築

コードカバレッジの計算などの有名な手法は既に様々なソフトウェアに組み込まれているが，動的解析手法の研究を行う場合は，その目的に合った実行トレースの取得が必要である．本章では，動的解析の研究において重要な要素となる，実行トレースの取得方法とその注意点について述べる．

5.1 実行トレースを記録するためのツール

Java を解析対象とした場合に利用可能なツールには，以下のようなものがある．

jdb デバッガ jdb は，JDK に標準で付属しているコマンドラインデバッガである．小さなプログ

```
jdb の初期化中です...
> stop in LeapYear.main
ブレークポイント LeapYear.main を保留しています。
クラスがロードされた後に設定されます。
> run
LeapYear 2100 を実行します
uncaught java.lang.Throwable を設定しました
保留した uncaught java.lang.Throwable を設定しました
>
VM が起動しました: 保留した ブレークポイント
LeapYear.main を設定しました

ブレークポイントのヒット: "スレッド=main", LeapYear.main(),
line=3 bci=0

main[1] monitor next
main[1] next
>
ステップ完了: "スレッド=main", LeapYear.main(), line=5 bci=10
> >
ステップ完了: "スレッド=main", LeapYear.main(), line=6 bci=16
> >
ステップ完了: "スレッド=main", LeapYear.main(), line=7 bci=23
> >
ステップ完了: "スレッド=main", LeapYear.main(), line=8 bci=32
> >
ステップ完了: "スレッド=main", LeapYear.main(), line=18 bci=53
> > yes
ステップ完了: "スレッド=main", LeapYear.main(), line=19 bci=60
> >
アプリケーションは終了しました
```

図 8 jdb を用いたステップ実行の自動反復

ラムの実行に対して実行トレースを書き出すのであれば，次のような手順で，簡単に実行することができる．

まず，jdb を実行する．これは `java` コマンドで Java プログラムを実行する場合と同様で，たとえば `jdb LeapYear 2100` のように入力する．jdb が起動した時点ではプログラムはまだ実行されていないので，次のように `main` の開始直後に停止するようブレークポイントを設定し，プログラムの実行を指示する．

```
stop in LeapYear.main
run
```

このコマンドを実行すると，プログラムの実行が開始され，`main` メソッドの先頭で実行が一時停止する．プログラムが一時停止している間は，プログラムの状態を観測するための様々なコマンドが実行できる．その 1 つである `monitor` コマンドは，プログラムが一時停止したときに自動的に実行したいコマンドを指定するもので，次のようにステップ実行命令である `next` コマンドを設定してから，`next` コマンドでプログラムの実行を開始すると，ステップ実行による停止に対してさらにステップ実行を行わせることができ，1 行ずつプログラムを実行することができる．

```
monitor next
next
```

jdb は，プログラムを一時停止したとき，停止したスレッド名，メソッド名，実行直前の命令の行番号，バイトコード上での位置をメッセージとして書き出す．例として，上記の処理を行ったときに jdb が出力したメッセージの列を図 8 に示す．“ステップ完了 `... line=8 ...`” という記述から，8 行目の命令 “`answer = "yes";`” に到達したことが示されており，プログラムの実行が意図とは異なる結果になっていることを知ることができる．jdb が出力する行番号の列を保存し，分析するツールを作成すれば，行カバレッジを計算することができる．この方式は，ステップ実行によってプログラムの一時停止，実行再開を繰り返すので，実行効率は非常に悪い．教育用の数十行のプログラムであれば問題なく実行できるが，一般的なプログラムを解析する用途には不向きである．

JVMTI JVMTI [22] は Java Virtual Machine Tool Interface の略称で，jdb を含めた様々なツールが Java 仮想マシンと通信するためのインタフェースである．観測用の「エージェント」は，JDK に付属している C 言語用のヘッダファイル `jvmti.h` を使って C 言語などで作成することができる動的リンクライブラリの一種で，JVMTI の仕様に従った特定の名前の関数を用意しておくと，プログラムの実行開始，メソッド呼び出し，スレッドの実行開始などの様々なイベントの通知を仮想マシンから受け取ることができる．

JVMTI は，Java 仮想マシンの外部からプログラムの動作を観測するため，スレッドの制御やメモリ管理など，Java プログラムの制御情報を獲得することができる．また，jdb を経由して Java 仮想マシンを操作する場合に比べれば高速であり，特定の目的に特化したツールを作成することができる．一方で，オブジェクトの参照や変数の値などを取得する場合には，逐一仮想マシン内部のデータ構造にアクセスすることになり，実行時に大きなオーバーヘッドとなることがある．

ASM ASM†3はバイトコードの読み書きを行うためのライブラリである．本稿で紹介した Cobertura も内部ではこれを使用している．ライブラリとしては，クラス構造がシンプルであること，Java Generics 情報を含むバイトコードも正しく扱えることが特徴である．実行トレースの記録を記述するための特別な支援が用意されているわけではないので，たとえばメソッド呼び出しとして `invokestatic` 命令を挿入するというように，Java バイトコードに関する知識が必要である．学習コストと引き換えに，柔軟さは高い．

バイトコード書き換えによって作成したコードは，対象プログラムの一部として実行されるため，JIT コンパイルなどの最適化の恩恵を受けることができ，JVMTI に比べると実行速度は高速となることが多い．プログラムのオブジェクトの状態や，変数の値にアクセスしやすいことが特徴である．一方で，スレッド間の同期制御など，解析対象のプログラムの記述の影響を受けやすく，プログラムの振舞いを破壊しないように慎重な実装が必要となる．また，バイトコード書き換え技術を用いてプログラムに処理を追加するようなフレームワークと併用する場合は，バイトコードの書き換え順序に注意する必要がある．

Javassist Javassist†4は，バイトコード書き換えツールの 1 つである．バイトコードとして書き加えたい処理の内容を Java のソースコードに近い形式で記述することができるため，ASM と比べると，バイトコードに詳しくない開発者でも容易に使うことができる．バイトコード書き換えというアプローチを採用しているので，その他の利点や弱点に関しては ASM と同様である．

AspectJ Java のアスペクト指向拡張である AspectJ †5は，メソッド呼び出しの記録など，ある一定の範囲に限っては，実行時情報の記録に関する処理を非常に簡単に記述することができる．記述したコードは，AspectJ の処理系によって Java のバイトコードへと組み込まれるため，その他の特徴は ASM や Javassist と同様である．

動的解析手法を自分で実装する場合，ツールの選び方が重要である．まず，実行速度を重視するのであれば，実行トレースの記録はプログラム本体に埋め込むほうがよい．そのため，バイトコード書き換えにあたる ASM，Javassist，AspectJ が優れている．このうち，メソッド単位など，粗い粒度の記録だけで十分なら，学習コストの観点から AspectJ が優れている．

スレッドの切り替わりやメモリ管理など，プログラムの外にある情報をできるだけ正確に解析したい場合や，プログラムを書きかえることによる処理の変化が気になる場合は，JVMTI が適している．jdb は，利用は容易であるがオーバーヘッドが大きいので，JVMTI で利用可能な情報を知るための学習用として使用し，本格的な解析を実施する場合は JVMTI に移行するべきである．

5.2 実行トレース記録時の注意

実行トレースは，プログラムをある特定の実行環境，特定の入力データに対して実行したときに獲得できる情報である．そのため，実行条件のわずかな違いから，実行トレースの結果が変わってしまうことに注意しなければならない．実行トレースに影響を与える要素を次に示す．

- ユーザによる操作．ユーザの操作を記録，再生するツールなどを使用しない限り，完全に同一の操作を繰り返すことはできない．マウス操作など

†3 `http://asm.ow2.org/`

†4 `http://www.csg.is.titech.ac.jp/~chiba/javassist/`

†5 `http://www.eclipse.org/aspectj/`

に対応して実行されるイベントハンドラは，異なる回数実行されることがある．

- マルチスレッド処理におけるスレッド間のインタラクション．スレッド実行のスケジュールは実行時に決定されるため，スレッド間での命令の実行順序は，毎回入れ替わることがある．
- タイマによって起動される処理の実行回数およびタイミング．実行トレースを取得する処理自体にも時間がかかるので，処理のタイムアウトの発生にも影響することがある．
- ファイルやデータベースなどの外部データ．ファイルに格納されているデータや，計算機の日付情報などの変化が，処理に影響を与えることがある．

動的プログラムスライシングのように，ただ 1 つの実行トレースを解析する手法では，同じ対象ソフトウェアを同じように実行したつもりでも，上記のような要因から異なる実行トレースが生成され，適用結果が異なったものになる可能性がある．そのため，評価実験を計画する際には，注意が必要である．プログラムの実行をできるだけ再現する方法については，デバッグ技法として Andreas Zeller が整理している [33]．なお，コードカバレッジの計算や，統計的デバッギングのように，複数回実行した結果を集計する手法では，実行に再現性がないことは問題にはならない．

6 まとめ

動的解析は，プログラムの実行を解析し，プログラムの正確な振舞いを開発者に提示する技術である．本稿ではテストとデバッグに役立つ代表的な技術を紹介したが，プログラム理解を目的とした研究も数多く実施されており，研究動向については Cornelissen らのサーベイ論文 [7] にまとめられている．

本稿で紹介したような動的解析の基本的な手法は，簡明なアイディアに基づいており，既存ツールを使いこなすことができれば，自分で開発することもそれほど難しくはない．本稿によって動的解析に興味を持った場合は，様々な研究論文を読み進めるとともに，簡単な解析手法を手元の環境で試作し，様々なプログラムを実行することで，動的解析の特徴を理解することが重要である．

参考文献

[1] Agrawal, H. and Horgan, J. R.: Dynamic Program Slicing, *SIGPLAN Notices*, Vol. 25, No. 6(1990), pp. 246–256.

[2] Akgul, T., Mooney III, V. J., and Pande, S.: A Fast Assembly Level Reverse Execution Method via Dynamic Slicing, *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*, May 2004, pp. 522–531.

[3] Bacon, D. F. and Sweeney, P. F.: Fast Static Analysis of C++ Virtual Function Calls, *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1996, pp. 324–341.

[4] ボーリス・バイザー: ソフトウェアテスト技法, 日経 BP 出版センター, February 1994.

[5] Binder, R. V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley Professional, October 1999.

[6] Briand, L. C., Labiche, Y., and Leduc, J.: Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software, *IEEE Transactions on Software Engineering*, Vol. 32, No. 9(2006), pp. 642–663.

[7] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R.: A Systematic Survey of Program Comprehension through Dynamic Analysis, *IEEE Transactions on Software Engineering*, Vol. 35, No. 5(2009), pp. 684–702.

[8] Dwyer, M. B., Kinneer, A., and Elbaum, S.: Adaptive Online Program Analysis, *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, May 2007, pp. 220–229.

[9] Edwards, D., Wilde, N., Simmons, S., and Golden, E.: Instrumenting time-sensitive software for feature location, *Proceedings of the 17th IEEE International Conference on Program Comprehension*, May 2009, pp. 130–137.

[10] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution, *IEEE Transactions on Software Engineering*, Vol. 27, No. 2(2001), pp. 99–123.

[11] Gabel, M. and Su, Z.: Online Inference and Enforcement of Temporal Properties, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp. 15–24.

[12] Gosling, J., Joy, B., Steele, G., and Bracha, G.: *Java Language Specification, The 3rd Edition*, Prentice Hall, June 2005.

[13] Jones, J. A., Harrold, M. J., and Stasko, J.: Visualization of Test Information to Assist Fault Localization, *Proceedings of the 24th ACM/IEEE International Conference on Software Engineering*,

May 2002, pp. 467–477.

[14] Joshi, S. and Orso, A.: SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions, *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, October 2007, pp. 234–243.

[15] Ko, A. J. and Myers, B. A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, May 2008, pp. 301–310.

[16] Lewis, B.: Debugging Backwards in Time, *Proceedings of the 5th International Workshop on Automated Debugging*, September 2003.

[17] Lo, D. and Maoz, S.: Mining Scenario-Based Triggers and Effects, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, September 2008, pp. 109–118.

[18] Lo, D., Ramalingam, G., Ranganath, V. P., and Vaswani, K.: Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation, *Proceedings of the 16th Working Conference on Reverse Engineering*, October 2009, pp. 62–71.

[19] Ngo, M. N. and Tan, H. B. K.: Detecting Large Number of Infeasible Paths through Recognizing Their Patterns, *Proceedings of the the 6th Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2007, pp. 215–224.

[20] Park, S., Vuduc, R. W., and Harrold, M. J.: Falcon: Fault Localization in Concurrent Programs, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp. 245–254.

[21] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J.: Visualizing the Execution of Java Programs, *Revised Lectures on Software Visualization, International Seminar*, 2001, pp. 151–162.

[22] Prasad, C. K., Ramchandani, R., Rao, G., and Levesque, K.: Creating a Debugging and Profiling Agent with JVMTI, June 2004.

[23] Quante, J. and Koschke, R.: Dynamic Object Process Graphs, *Journal of Systems and Software*, Vol. 81(2008), pp. 481–501.

[24] Reiss, S. P.: Dynamic Detection and Visualization of Software Phases, *Proceedings of the International Workshop on Dynamic Analysis*, 2005, pp. 1–6.

[25] 櫻井孝平, 増原英彦, 古宮誠一: Traceglasses: 欠陥の効率よい発見手法を実現するトレースに基づくデバッガ, 情報処理学会論文誌 *プログラミング (PRO)*, Vol. 3, No. 3(2010), pp. 1–17.

[26] Santelices, R. and Harrold, M. J.: Efficiently Monitoring Data-flow Test Coverage, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, November 2007, pp. 343–352.

[27] 下村隆夫: プログラムスライシング技術と応用, 共立出版, July 1995.

[28] Sundaresan, V., Hendren, L., and Razafimahefa, C.: Practical Virtual Method Call Resolution for Java, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000, pp. 264–280.

[29] Systa, T.: Understanding the Behavior of Java Programs, *Proceedings of the 7th Working Conference on Reverse Engineering*, November 2000, pp. 214–223.

[30] Tallam, S., Tian, C., and Gupta, R.: Dynamic Slicing of Multithreaded Programs for Race Detection, *Proceedings of the 24th IEEE International Conference on Software Maintenance*, September 2008, pp. 97–106.

[31] Wang, X., Cheung, S. C., Chan, W. K., and Zhang, Z.: Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization, *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering*, May 2009, pp. 45–55.

[32] Watanabe, Y., Ishio, T., and Inoue, K.: Feature-level Phase Detection for Execution Trace Using Object Cache, *Proceedings of the International Workshop on Dynamic Analysis*, 2008, pp. 8–14.

[33] Zeller, A.: *Why Programs Fail: a Guide to Systematic Debugging*, Morgan Kaufmann, October 2005.

[34] Zhang, X., Tallam, S., and Gupta, R.: Dynamic Slicing Long Running Programs Through Execution Fast Forwarding, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2006, pp. 81–91.

[35] Zhang, Z., Chan, W. K., Tse, T. H., Jiang, B., and Wang, X.: Capturing Propagation of Infected Program States, *Proceedings of the 7th Joint Meeting of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2009, pp. 43–52.

[36] Zheng, A. X., Jordan, M. I., Liblit, B., and Aiken, A.: Statistical Debugging of Sampled Programs, *Advances in Neural Information Processing Systems 16*, Thrun, S., Saul, L., and Schölkopf, B.(eds.), MIT Press, Cambridge, MA, June 2004.