Javaプログラムを対象とした単体テスト可視化ツール “Jvis” の開発

松岡 慎吾[1]　喜多 義弘[,2]　片山 徹郎[,1]

概要：本稿では，ソフトウェア開発における単体テストの実施状況をリアルタイムに可視化することによって，テスト実施状況の理解および確認に費やす手間の削減を目的としている．目的達成のアプローチとして，Java プログラムを対象とした単体テスト可視化ツール “Jvis”(Tool for **J**ava programs to **vis**ualize unit testing) を開発した．Jvis は，テスト対象コードに対して，ステートメントカバレッジ (C0) とブランチカバレッジ (C1) に基づいた自動テストを実施し，現在のテスト実施状況をリアルタイムに提供する．これによって，テスト実施状況の理解および確認に費やす手間の削減を実現する．

Development of Unit Testing Visualization Tool “Jvis” for Java Programs

Matsuoka Shingo[1]　Kita Yoshihiro[,2]　Katayama Tetsuro[,1]

Abstract: This paper aims to reduce the effort spent on understanding and confirming the testing progress by visualizing the progress of unit testing in software development in real-time. As an approach to achieve the goal, an automatic unit testing and visualization tool “Jvis” (Tool for **J**ava programs to **vis**ualize unit testing) has been implemented. Jvis executes automated testing based on C0 (statement coverage) and C1 (branch coverage) for the test target program. And Jvis visualizes current progress of testing in real-time. Hence, Jvis can reduce the effort spent on understanding and confirming the testing progress.

1. はじめに

近年，ソフトウェア開発における上流工程の可視化手法 (例えば, UML, DFD, ERM) が提案されている [1], [2], [3]. これらの可視化手法は，自然言語で記述される設計書と比べて，ソフトウェアの構造の理解や開発者間の情報共有を容易にする．一方で，ソフトウェア開発における下流工程の可視化に関する研究については, 幾つかの報告 [4], [5], [6] があるものの，十分に研究が進んでいない領域である．一般に，自動化されたテストでは大量の実行結果が出力される．大量に羅列された実行結果や，自然言語で記述されたログから，テスト対象コードのどこをどれだけテストできたのかを瞬時に理解することは容易ではない．このことは，テスト技術者間の円滑な情報共有を難しくする．テスト実施状況の理解と共有に時間を費やすことは，テストの効率を低下させる 1 つの要因であると考える．

そこで本稿では，テスト実施状況の理解および確認に費やす手間の削減を目的として，Java プログラムを対象とした単体テスト可視化ツール “Jvis” (Tool for **J**ava programs to **vis**ualize unit testing) を開発する．Jvis は，テスト対象コードに対する単体テストを自動的に実施し，単体テストの実施状況をリアルタイムに可視化する．

Jvis の概要を，以下に示す．まず，テスト対象コードを元に，ブランチカバレッジ (C1) に基づいたテストの可視化に必要な命令文を追記した可視化用コードを生成し，Jvis 上に表示する．C1 とは，テスト対象コードの分岐をどれだけ実施したかを表す基準である [7]．次に，可視化用コードを元に，カバレッジ計測コードを生成する．カバレッジ計測コードとは，可視化用コードを元に，カバレッジの計測に必要な命令文の挿入や，命令文の書き換えを行ったコードである．次に，ステートメントカバレッジ (C0) および C1 に基づいた自動テストを実施する．C0 とは，テスト対象コードの命令文をどれだけ実施したかを表す基準であ

[1] 宮崎大学
University of Miyazaki
[2] 神奈川工科大学
Kanagawa Institute of Technology

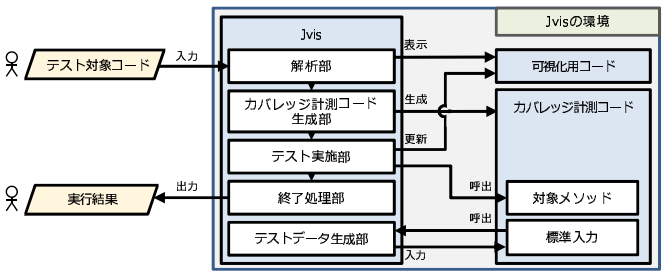

図 1 Jvis の全体構成
Fig. 1 Configuration of Jvis

る [7]．Jvis は，テストの実施中，可視化用コードのハイライトや，命令文毎の実行回数の表示によって，テスト実施状況をリアルタイムに可視化する．最後に，テスト実施結果を HTML(HyperText Markup Language)[8] 形式でファイルに出力する．

以下，本稿の構成は次のとおりである．2 章では，Jvis の機能と実装方針について述べる．3 章では，Jvis の適用例を示す．4 章では，Jvis についての考察を行う．5 章では，本稿のまとめと今後の課題について述べる．

2. Jvis の機能と実装方針

この章では，Jvis の機能および実装方針について説明する．

Jvis は，大きく 3 つの機能を持っている．

- テストデータの自動生成およびテストの自動実施: テストデータを自動生成し，テスト対象コードに入力することによってテストを自動実施する．
- C0 および C1 の計測: テストを実施することによって，テスト対象コードの C0 および C1 を計測する．
- テスト実施状況の可視化: テストの実施によって網羅した命令文をハイライトし，テスト実施状況をリアルタイムに可視化する．

Jvis の全体構成を，図 1 に示す．Jvis は，解析部，カバレッジ計測コード生成部，テスト実施部，テストデータ生成部，終了処理部の 5 つで構成する．各部の詳細を，以下に示す．

2.1 解析部

解析部では，ユーザが選択したテスト対象コードを読み込む．C1 に基づいたテストの実施状況を可視化するため，読み込んだテスト対象コードに対して正規表現 [9] を用いた解析を行い，else 文を持たない if 文と，default 文を持たない switch case 文に対して，それぞれ else 文，default 文を追記する．本稿では，C1 に基づいたテストの実施状況を可視化するために追記を行ったこのコードを，可視化用コードと呼ぶ．ここで追記する else 文，default 文は，それぞれ else{}，default: とし，内部処理は付加しないため，テスト対象コードの処理内容に変更は生じない．また，テスト実施に必要なクラス名の抽出，および引数を含む各メソッド名の抽出を行う．最後に，可視化用コードを，ソースコード表示ラベルに表示する．

2.2 カバレッジ計測コード生成部

カバレッジ計測コード生成部では，可視化用コードを元に，カバレッジの計測に必要な命令文の挿入や，命令文の書き換えを行ったカバレッジ計測コードを生成する．カバレッジ計測コードの生成における主なパターンと，命令文の挿入および書き換えの処理を，表 1 に示す．ここで，命令文および分岐の網羅を検出するため，命令文および分岐に対して，それぞれの網羅回数を記憶するカウンタを挿入する．また，ユーザの入力を求める標準入力命令を，メソッド名を元に，対応するテストデータ生成部の呼び出し命令に書き換える．

2.3 テスト実施部

テスト実施部では，ユーザが選択するメソッドを起点に，カバレッジ計測コードを実行する．引数を持つメソッドを呼び出す場合は，引数の型に対応したテストデータを生成し，対象メソッドに対して入力する．現在，メソッドに対する入力は，現在，int 型，short 型，long 型，float 型，double 型，byte 型，boolean 型，String 型，そして，これら 8 つの型の配列に対応している．

また，カバレッジ計測コードの実行によって取得した命令文および分岐の網羅状況を元に，ソースコード表示ラベルに表示する可視化用コード 1 行毎を随時ハイライトする．さらに，カバレッジ計測コードを実行する度に，ソースコード表示ラベルに表示する各命令文の実行回数を，カウンタの値を元に随時更新する．取得した命令文および分岐の網羅状況を元に C0 および C1 を算出し，C1 の値が 100%に達する，または，テスト停止のため Jvis に設置している停止ボタンを押すまで，テスト実施部はカバレッジ計測コードの実行を繰り返す．

2.4 テストデータ生成部

テストデータ生成部では，ユーザの入力を求める標準入力命令に代わって，無作為にテストデータを生成する．カバレッジ計測コード生成部によってテストデータ生成部の呼び出し命令に書き換えることができる標準入力命令と，生成データの範囲には制限がある．なお，現在，Jvis が対応する生成データの型は，次のとおりである; int 型，short 型，long 型，float 型，double 型，byte 型，boolean 型，String 型．

2.5 終了処理部

カバレッジ計測コードの C1 の値が 100%に達した場合，または停止ボタンを押した場合，テスト実施部から終了処

表 1 カバレッジ計測コード生成部の主な処理一覧
Table 1 Processes by coverage instrumented code generator

処理内容	パターン	パターンマッチ後の処理
標準入力命令の書き換え	.*System.in.read¥¥(.*¥¥);.* など	System.in.read([変数名]); → Jvis.read([変数名]);
命令文に対するカウンタの挿入	すべての命令文 ※ 命令文を挿入することによってエラーが発生しないすべての命令文が対象。解析部で追記した命令文は対象外。エラーが発生する文法箇所はパターンマッチで検索し、別処理を行う(下項: エラー回避を参照)。	[命令文]; → [命令文];Jvis.statement[行数]++;
命令文に対するカウンタの挿入(エラー回避)	.*return.*;.* など ※ return文やbreak文、if文を閉じる } の直後など、到達不可能や文法のエラーを理由に、命令文の挿入によってエラーが発生する箇所が対象。	return [戻り値]; → Jvis.statement[行数]++;return [戻り値]; if(条件式){ … } → if(条件式){ … } //カウンタを挿入しない
分岐命令に対するカウンタの挿入	.*if¥¥s*¥¥(.*¥¥).* など	if(条件式){ → if(条件式){Jvis.branch[行数]++;
戻り値の記録	.*return.*;.*	return [戻り値]; → Jvis.returnStorage[実行回数][要素数]=Jvis.visualizationCode.get([行数]);return [戻り値];

```
 3  public static int Inventory_check(int number, int amount){
 4    String[] Inventory = null;
 5    for(int i=0;i<Inventory_load().size();i++){
 6      Inventory=((String)Inventory_load().get(i)).split(",",0);
 7      if(Inventory[0].equals(String.valueOf(number))){
 8        if(Integer.parseInt(Inventory[2])-amount<0){
 9          return -1;                    //在庫不足
10        }
11        else{
12          return 1;                     //在庫充足
13        }
14      }
15    }
16    return 0;                           //商品コードなし
17  }
```

図 2 簡易在庫管理プログラムの一部
Fig. 2 Part of the inventory management program

理部に移行する．終了処理部では，実行結果を HTML 形式のファイルに出力する．実行結果として，次の情報をファイルに書き出す; (1) 可視化用コードの網羅状況と，命令文毎の実行回数, (2)C0 および C1 の値, (3) 総実行回数, (4) 各メソッドと，各標準入力に使用したテストデータ．(1)〜(3) については同一ページ (index.html) に，(4) については [メソッド名または標準入力命令 (行番号).html] の名前でそれぞれ出力する．

3. 適用例

本研究で開発を行った Jvis が正しく動作することを検証するため，Java プログラム「簡易在庫管理プログラム」を Jvis に適用した．

図 2 に，「簡易在庫管理プログラム」の一部である在庫確認メソッド (Inventory_check) を示す．簡易在庫管理プログラムをテスト対象コードとして Jvis に与え，カバレッジ計測コード生成部が生成したカバレッジ計測コードの一部 (在庫確認メソッド) を，図 3 に示す．以下に，図 2 のプログラムを元に，図 3 のカバレッジ計測コードを生成する処理について説明する．なお，図 2，図 3 の左にある番号は行番号である．

- 命令文の網羅状況を取得するため，挿入によってエラーが発生しないすべての命令文の直後にカウンタ *statement*[行番号]++; を挿入する (図 3 の 3〜8, 11 行目).
- 命令文直後の挿入ではエラーが発生するため，命令文の直前にカウンタ *statement*[行番号]++; を挿入する (9, 12, 17 行目).
- 分岐の網羅状況を取得するため，分岐命令の直後にカウンタ *branch*[行番号]++; を挿入する (7, 8, 11, 15 行目).
- return 文の直前に，return 文の格納を行う代入文を挿入する (9, 12, 17 行目).

図 4 は，簡易在庫管理プログラムの Inventory_check メソッドを起点にテストを実施し，終了した後の Jvis の外観である．今回適用した簡易在庫管理プログラムは，C1 の値を 100%満たすことができたため，カバレッジ計測コードの実行は自動的に停止した．すべての命令文，および解析部で追記を行った else 文を，若草色にハイライトしていることから，すべての命令文および分岐方向を網羅できたことを確認した．

図 5 に，ウェブブラウザで開いた簡易在庫管理プログラムの実行結果の一部を示す．index.html の各命令文左側の数字は, 命令文毎の実行回数を表す．Inventory_check(3).html には，Inventory_check メソッドに与えた引数と，引数に対応する戻り値を出力している．

次に，在庫確認メソッド内の for 文 (図 2: 5 行目) の条件式が誤った状態で適用する．下記のように，不等号の向きを誤って記述した場合を仮定している．

for(int i=0;i >Inventory_load().size();i++)

for 文内のコードはデッドコードとなり，在庫の充足・不足，商品コードの有無にかかわらず，Inventory_check メソッドは戻り値として 0 を返す状態である．図 6 は，このデッドコードを含んだ簡易在庫管理プログラムのカバレッジ計測コードを，約 3 分間実行し続けた Jvis の外観である．デッドコードをハイライトせず，現在の C1 の値を示すプログレスバーは，33%で頭打ちになっている．デッドコードを含んだ簡易在庫管理プログラム内に 6 個ある分岐方向に対して，網羅した分岐方向が 2 個 (2 ÷ 6　0.333) であるため，C1 の値を正しく取得している．実行結果を確認したところ，C0 は 83%(57 ÷ 68　0.838; for 文内のデッドコードが 11 行，簡易在庫管理プログラム全体のコー

```
 3  public static int Inventory_check(int number, int amount){Jvis.statement[3]++;
 4    String[] Inventory = null; Jvis.statement[4]++;
 5    for(int i=0;i<Inventory_load().size();i++){Jvis.statement[5]++;
 6      Inventory=((String)Inventory_load().get(i)).split(",",0); Jvis.statement[6]++;
 7      if(Inventory[0].equals(String.valueOf(number))){Jvis.statement[7]++;Jvis.branch[7]++;
 8        if(Integer.parseInt(Inventory[2])-amount<0){Jvis.statement[8]++; Jvis.branch[8]++;
 9          Jvis.statement[9]++;Jvis.returnStorage[Jvis.NumOfRuns][Jvis.NumOfEle++]=Jvis.visualizationCode.get(9);return -1;
10        }
11        else{Jvis.statement[11]++; Jvis.branch[11]++;
12          Jvis.statement[12]++; Jvis.returnStorage[Jvis.NumOfRuns][Jvis.NumOfEle++]=Jvis.visualizationCode.get(12);return 1;
13        }
14      }
15      else{Jvis.branch[15]++;}
16    }
17     Jvis.statement[17]++; Jvis.returnStorage[Jvis.NumOfRuns][Jvis.NumOfEle++]=Jvis.visualizationCode.get(17);return 0;
18  }
```

図 3 簡易在庫管理プログラムのカバレッジ計測コードの一部

Fig. 3 Part of coverage instrumented code of the inventory management program

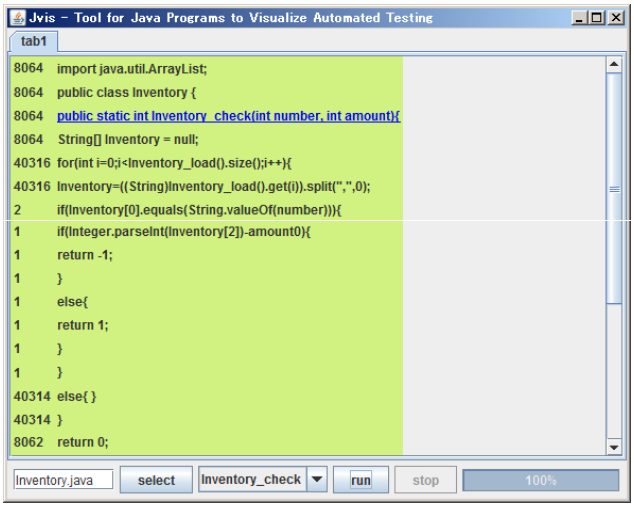

図 4 簡易在庫管理プログラム適用後の Jvis の外観

Fig. 4 An overview of Jvis after application of the inventory management program

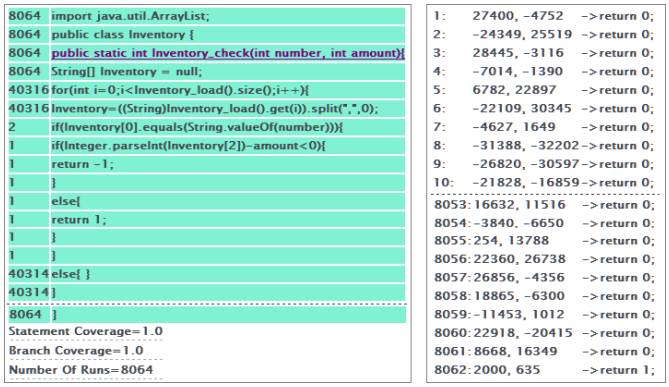

図 5 簡易在庫管理プログラムの実行結果の一部
(左: index.html, 右: Inventory_check(3).html)

Fig. 5 Part of the execution result of the inventory management program
(left: index.html, right: Inventory_check(3).html)

ド行数が 68 行) を示していた．また，登録のある商品コードを引数として入力しているにもかかわらず，すべての戻り値が return 0; であった．

以上，簡易在庫管理プログラムを適用した結果，実装方針どおりにカバレッジ計測コードの生成ができており，Jvis が正しく動作することを確認した．また，ソースコード表示ラベルに表示する可視化用コードをハイライトし，各命令文毎の実行回数を示すことによって，テスト実施状況をリアルタイムに正しく提示できることを確認した．

4. 考察

本稿では，ソフトウェア開発における単体テストの実施状況をリアルタイムに可視化することによる，テスト実施状況の理解および確認に費やす手間の削減を目的として，単体テスト可視化ツール “Jvis” を開発した．Jvis は，テスト対象コードに対してカバレッジに基づいた自動テストを実施する．カバレッジ計測コードを生成することによって，命令文および分岐の網羅状況を取得し，テスト実施状況のリアルタイムな可視化を実現する．この可視化によって，C0 および C1 を基準としたテスト実施状況の直感的な理解を支援する．また，リアルタイムな可視化によって，現在のテスト実施状況を，テスト実施中に確認することが可能である．C1 はテストの終了基準として用い，テスト対象コードが C1 の値を 100%満たすと自動的にテストを終了する．テスト実施後，取得した実行結果を HTML 形式でファイルに出力する．

Jvis は，テスト手法としてランダムテストを採用している．手動によるランダムテストは，同値分割や境界値分析といったドメインベースのテストに比べて効率が悪い [10]．しかし，自動テストとしてランダムテストを用いた場合，短時間で大量のテストデータを入力することができる．Jvis を使用し，テストデータの入力によるプログラムの網羅状況を可視化することによって，自動テストによる大量のテストデータが，テスト対象コード中のどこをどれだけテストしたのかを直感的に理解することができる．さらに，テスト実施状況をリアルタイムに可視化しているため，一般のテスト結果報告書では確認できないテスト対象コードの網羅状況を，動的に見ることができる．なお，テストデータをランダムに生成するため，テスト対象コードによっては C1 を 100%満たすまでに時間を要する場合もあるが，テ

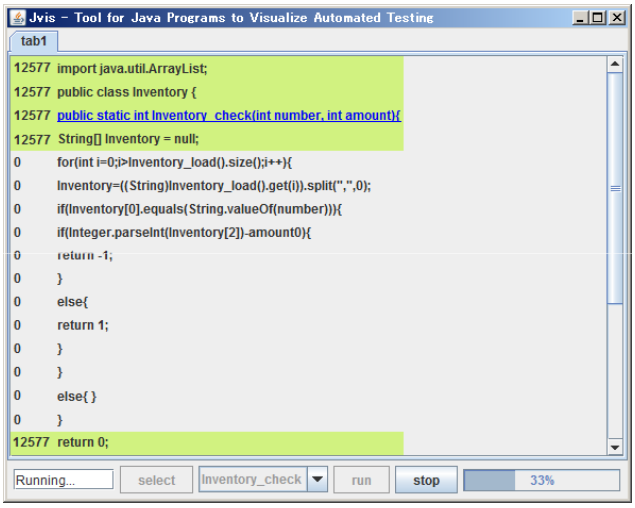

図 6 デッドコードを含んだ簡易在庫管理プログラムのカバレッジ計測コードを約 3 分間実行し続けた Jvis の外観

Fig. 6 Demonstration of Jvis continued to execute the inventory management program including dead code about 3 minutes

ストの停止基準として C1 を用いていることから，この基準に応じたテストを実施することができる．

現在，テスト自動化を支援する様々なテストツールが提案されている [11], [12], [13], [14], [15]．テストケースの自動生成，静的解析によるコード分析，カバレッジの測定など，各ツールの機能は様々だが，テスト実施状況を可視化するツールは少ない．また，Jvis のように，テスト実施状況をリアルタイムに提示するツールは公開されていない．

テスト実施状況を可視化するツールとして，jcoverage[16] が挙げられる．jcoverage は，テスト実施によるテスト対象コードの網羅状況を可視化し，レポートとして出力する．

jcoverage は，テスト対象コード中の網羅した命令文を緑色に，網羅していない命令文を赤色に可視化したテスト実施状況と，C0 および C1 の値を，カバレッジレポートとして出力する．C1 が値として出力される一方，テスト対象コードを元に可視化処理を施したテスト実施状況からは，分岐の網羅状況を確認できない．例えば，テスト対象コードが else 文を持たない if 文を含み，テスト実施によって if 文の条件式を満たした場合，この if 文の処理が緑色にハイライトされ，可視化される．しかし，記述されていない else 方向の分岐を網羅したか否かという情報は，カバレッジレポートを見ただけでは確認することができない．Jvis では，C1 に基づいたテストの可視化に必要な命令文を追記した可視化用コードを生成し，これをハイライトすることによってテスト実施状況を可視化している．このため，C1 の値だけでなく，すべての分岐方向を網羅したか否か，C1 に基づいたテストの実施状況を，直観的に理解することができる．

jcoverage を用いた可視化では，すべてのテストが実施された後，テスト対象コード中の各命令文の実行回数を含めた最終的な網羅状況がレポートとして出力されるため，テスト実施中の様子を確認できない．例えば，テスト対象コードがエラーを含む場合，テスト終了後に出力されるカバレッジレポートを確認し，エラー箇所を特定することになる．Jvis は，命令文および分岐の網羅状況を取得するカウンタを挿入したカバレッジ計測コードを生成することによって，テスト実施における網羅状況をリアルタイムに提示する．このため，テスト終了を待つことなく，エラーの検出を知ることができるので，テスト終了後にカバレッジレポートを確認し，エラー箇所を特定する作業が必要なくなり，手間の削減に繋がる．テスト対象コードおよびテストコードが大きくなり，テスト実施に要する時間が長くなるほど，手間削減の効果は大きくなると考えられる．

Java 言語には，Java 仮想マシンで動作するアプリケーションの状態検査や実行制御の機能を提供する Java Virtual Machine Tool Interface(JVMTI) というインタフェースが存在する [17]．この JVMTI を利用することによって，実行中の Java アプリケーションの監視や，ログファイルを使用した実行状態のトレース解析が可能である．JVMTI によるトレース解析では，Jvis における命令文の網羅状況取得を目的としたカウンタ (いわゆるプローブ) の挿入を必要としないという利点がある．しかし，JVMTI の提供する機能を利用する場合，アプリケーションプログラム内で発生するイベントから様々な状態の発生通知を受け取るための JVMTI エージェントを用意する必要がある．JVMTI エージェントを作成するためには，JVMTI の仕組みや JVMTI エージェントの記述方法など，特有の知識の習得が必要であり，また作成するための余計な手間がかかる．Jvis は，テスト対象コードを指定するだけで容易に自動実行および可視化を行うことが可能であり，これは Jvis におけるメリットの一つであると考える．

以上から，Jvis を用いたテストを実施することによって，テスト実施中に，現在のテスト実施状況をリアルタイムに確認でき，テスト対象コードの異常に気付くこともできる．また，Jvis の使用による前準備および前知識は必要なく，容易に利用することができる．これによって，テスト実施における手間を削減できることが見込まれる．

Jvis を用いた可視化では，短時間で大量のテストデータを生成する一方，各々のテストデータに着目した実行経路をトレースする目的には向いていない．すなわち，テスト対象コード中の “どこをどれだけテストしたのか” は一目で理解することができるが，“どのテストデータがどこをテストしたのか” という情報は可視化できていない．テスト実施後に出力する各テストデータに対して，実行経路の情報を付加し，テストデータ毎の網羅状況を可視化できるように改善する必要がある．

5. おわりに

本稿では，ソフトウェア開発における単体テストの実施状況をリアルタイムに可視化することによる，テスト実施状況の理解および確認に費やす手間の削減を目的として，Java プログラムのための単体テスト可視化ツール “Jvis” を開発した．

Jvis は，カバレッジ計測コードを生成し，カバレッジ計測コードにテストデータを入力することによって単体テストを自動実施する．また，カバレッジ計測コードを生成することによって，命令文および分岐の網羅状況を取得し，現在のテスト実施状況をリアルタイムに提示する．テスト終了後，取得した実行結果は，HTML 形式でファイルに出力する．Jvis を使用した自動テストを実施することによって，テスト対象コード中のどこをどれだけテストしたのかを直観的に理解することができる．また，リアルタイムな可視化によって，現在のテスト実施状況を，テスト実施中に確認することが可能である．

以上から，本稿で開発した Jvis を使用することによって，現在のテスト実施状況をリアルタイムに確認でき，単体テスト工程におけるテスト実施状況の理解および確認に費やす手間の削減が見込まれる．

以下に，今後の課題を挙げる．

- 可視性の向上: 本提案手法では，可視化用コードをハイライトすることによって，C1 に基づいたテストの可視化を実現している．分岐や繰り返し処理における流れの可視性を向上するため，フローチャートなど，よりグラフィカルな可視化手法を用いることを検討する必要がある．
- 可視化の強化: 現バージョンでは，C0 および C1 カバレッジのみを可視化している．“どのテストデータがテスト対象コード中のどこをテストしたのか” という情報は可視化していないため，テストデータ毎の流れや振る舞いを理解することは難しい．このため，各テストデータに着目し，テストデータ毎の網羅状況を可視化できるような改善を考えている．
- 実用性の向上: 現状では，自動生成する型や生成データの範囲，ユーザによる入力の自動化に対応するメソッドの制限など，適用の制約がある．特に，Java 言語によるプログラミングでは，プリミティブ型以外のデータ型を用いる頻度が極めて高い．また，テスト対象コードの解析に正規表現を用いたパターンマッチを行っているが，これではプログラムの多様な記述に対応できない．様々な構造，振る舞いを持つプログラムを適用し，問題点を洗い出した上で，生成データの拡張や構文解析器の導入など，実用性の向上を行う必要がある．

参考文献

[1] Chen, P. P.: The Entity - Relationship Model - Towards a Unified View of Data, ACM Transactions on database Systems, Vol.1, No.1, (1976).

[2] DeMarco, T.: Structured Analysis and System Specification, Yourdon Press, (1978).

[3] Booch, G., Rumbaugh, J. and Jacobson, I.: The Unified Modeling Language User Guide (2nd Edition), Addison Wesley Object Technology Series, (2005).

[4] Jones, J. A., Harrold, M. J. and Stasko, J.: Visualization of Test Information to Assist Fault Localization, Proc. 24th International Conference on Software Engineering, pp. 467-477, (2001).

[5] Cleanscape Software International: xATAC: Test Effectiveness Measurement Tool (online), available from ⟨http://legacy.cleanscape.net/products/testwise/tools_atac.html⟩ (accessed 2012-05-01).

[6] Kita, Y., Katayama, T., Tomita, S.: Proposal of Execution Paths Indication Method for Integration Testing by Using an Automatic Visualization Tool ‘Avis,’ 5th World Congress for Software Quality, J-13, (2011).

[7] Japan Software Testing Qualifications Board: テスト技術者資格制度 Foundation Level シラバス 日本語版 Version 2011.J01 (online), available from ⟨http://jstqb.jp/dl/JSTQB-Syllabus.Foundation_Version2011.J01.pdf⟩ (accessed 2012-05-10).

[8] Kennedy, B. and Musciano, C.: HTML; The Definitive Guide (3rd edition), O’Reilly Media, (1998).

[9] Friedl, J. E. F.: Mastering Regular Expressions (3rd edition), O’Reilly Media, (2006).

[10] Duran, J. W., Ntafos, S. C.: An Evaluation of Random Testing, IEEE Transactions on Software Engineering, Vol. SE-10, Issue. 4, pp. 438-444, (1984).

[11] Godefroid, P., Klarlund, N. and Sen, K.: DART: Directed Automated Random Testing, ACM SIGPLAN Notices, 40(6), pp. 213-223, (2005).

[12] Sen, K., Marinov, D. and Agha, G.: CUTE: A Concolic Unit Testing Engine for C, Proc. 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 263-272, (2005).

[13] GAIO TECHNOLOGY Co., LTD.: CoverageMaster winAMS (online), ⟨http://www.gaio.com/product/dev_tools/pdt07_winams.html⟩ (accessed 2012-05-02).

[14] Parasoft Co.: JTest (online), ⟨http://www.parasoft.com/jsp/products/jtest.jsp⟩ (accessed 2012-05-02).

[15] Agitar Technologies Inc.: AgitarOne (online), ⟨http://www.agitar.com/solutions/products/agitarone.html⟩ (accessed 2012-05-02).

[16] 野村総合研究所: jcoverage(Cobertura) 利用ガイド (online), available from ⟨http://works.nri.co.jp/service/pdf/jcoverage_cobertura.pdf⟩ (accessed 2012-04-26).

[17] Sun Microsystems, Inc.: Java Virtual Machine Tool Interface (JVMTI) リファレンス (online), available from ⟨http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jvmti/index.html⟩ (accessed 2012-07-31).