CDIツール開発におけるテストケース生成に関する研究

2008MI040 檜垣 佑輔　　2008MI068 石原 広大
指導教員　張 漢明

1　はじめに

コードインスペクションとは，コードを実行させる前にソフトウェア開発者が意図していない欠陥を発見する技術の総称である．本研究室では，Java ソースコードに対するコードインスペクションとツールして JavaCode Inspector[2]（以下 JCI) を開発している．現在 JCI では，Java の構文知識に基づいて新しい検査項目を作成することのできる環境の提供が試みられている．実現方法として，検査項目を表現する状態遷移機械を定義し，抽象構文木やその他の解析結果であるグラフを探索することでイベントを発生させ，そのイベントに基づいて状態を遷移させる方法が試みられている．その際，その検査項目の品質を保証するには，判定結果に違いがでるような部分的に異なる記述を持つ，類似したソースコードが多数必要となる．

本研究では，JCI 開発におけるテストケース作成の支援を目的として，状態遷移機械として記述された検査項目に対して，テストケースを自動生成する手法を提案する．先行研究 [4] で提案されている，モデルベースドテスト [1] を現在のテストプロセスに適用し，状態遷移機械からテストケースを自動生成する．具体的には，状態遷移についての経路の候補を計算し，その経路に基づいて Java のコードを組み立てていく方法を考えた．このような方法でテストケースを自動生成することで，多数の類似したソースコードを予想される結果とともに自動生成することができ，テストケース作成にかかるコストを少なくすることができる．本研究では，先行研究の不十分な点や相違点を考察したうえで，それらの点についての解決方法を示すとともに，テストケースの自動生成ツールの概略を示す．

2　背景技術

2.1　JCI

ソースコードを静的に検査するツールとして，不具合を起こしやすい記述やコーディング規約の違反箇所を検出するコードインスペクションツール（以下 CDI ツール）が存在する．JCI は Java ソースコードに対して，コーディングスタイルや構文規則などの観点から静的に検査をおこない，不具合が生じる可能性のある部分を指摘する CDI ツールである．現在 JCI が提供する検査項目は 36 種類あるが，利用者の要求によって検査の内容は多様に変化することから，JCI は検査機能の拡張やカスタマイズに対して柔軟に対応できるように設計されている．また，検査機能の追加の際には，新しく作成した検査項目の品質を確保するために，必要十分なテストケースを用意し，ソフトウェアテストを行なう必要がある．

2.2　モデルベースドテスト

モデルベースドテストとは，ソフトウェアテスト手法の一つで，テスト対象を記述したテスト設計モデルを用いてテストケースを設計する技術の総称である．

テスト対象の特定の性質を表現したモデルを基にテストケースの作成や，期待結果の表示などの一般的なテストの作業を行なう．モデルベースドテストは一般的にブラックボックステストとして実施される．テスト設計モデルは制御フローモデルや状態遷移モデルなど様々な形で表現することができ，ソフトウェア分析・設計の際に定義したモデルを使用することも可能である．

2.3　先行研究

先行研究では，JCI のテストプロセスにモデルベースドテストを適応させ，テストケースを自動生成する方法が提案されている．モデルには検査処理を表現するアクティビティ図を使用した．また，アクティビティ図の分岐にパスを割り振り，そのパスの組合せに対しソースコードを対応付けることで様々なテストケースが生成可能であることを示した．しかし，アクティビティ図の作成法・アクティビティ図の辿り方などに関しては定義されていない．

3　仕様記述の定義

新たな JCI の検査方法として，検査項目の仕様を状態遷移で表し，状態遷移機械を用いて検査項目の処理を行なうことが提案されている．抽象構文木の探索によって実現可能な検査項目を対象とした場合，以下のような手順で検査を行なう．

1. ソースコードに対して構文解析を行ない，抽象構文木を作成する
2. 作成した抽象構文木に対して深さ優先探索を行ない，if 文や for 文などの構文要素の開始や終了のイベントを発生させる
3. 発生させたイベントを状態遷移機械に渡し，状態を遷移させる，その際警告状態に遷移したら警告する
4. 特定の構文要素が入れ子になっていた場合，外側がどの部分を分析していたかをスタックに保存．内側の構文要素の分析が終わったときに，どこに復帰するかをスタックの情報から取得する
5. 全てのイベントを状態遷移機械が受け取ったら検査を終了する

検査対象の例として，既存の「if 文の条件式内に後置式が存在する場合警告する」検査項目の状態遷移機械を例として**図 1** に示す．この検査項目の仕様は，if 文の条

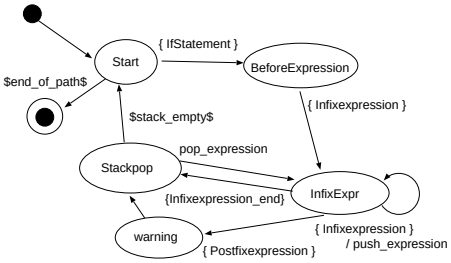

図 1　if 文の条件式内に後置式が存在する場合警告する

件式にインクリメント演算子やデクリメント演算子の後置式が存在する場合に警告を行なう．

この検査項目の処理の流れを以下に示す．

1. 検査対象のソースコード内に if 文が存在したら，**Start** から **BeforeExpression** に遷移
2. **BeforeExpression** では if 文に式が存在したら，**InfixExpr** に遷移
3. **InfixExpr** では if 文の式が後置式であれば，**warning** に遷移し警告，式が複数あればスタックに push を行ない **InfixExpr** に再帰，着目する式がなければ **Stackpop** に遷移
4. **Stackpop** ではスタック内に式が格納されている場合は，式を pop し **InfixExpr** に遷移，スタックが空の場合は，**Start** に遷移
5. **Start** で検査対象のソースコード内に存在する全ての if 文に関して，検査を行なった場合に終了状態に遷移し検査を終了

4　モデルベースドテストの適用

仕様書を基にテストケースを作成する際，テストケース作成者は検査の仕様から考えられる全ての状態を考慮してテストケースを作成する．しかし，JCI の検査処理の品質を保証するには構文要素の種類，制御の流れ，データの流れの組合せを考慮し，テストケースを作成する必要がある．これらの組合せは多数存在するのでテスト担当者は多量のテストケースを作成しなければならない．また，要求が変更された際には要求とともに仕様書も変更されるので，仕様書を基に作成されたテストケースも変更する必要があり，テストケース作成をやり直さなければならない．現在これらは手作業で行なっており，テスト担当者の労力が大きい．この問題点に対して検査処理を表現したモデルを基に，テストケースを自動生成することでテストケース作成の支援を図ることが先行研究で行なわれた．先行研究ではモデルとしてアクティビティ図を使用していた．例として先行研究で使用していた，「無限ループを検出する」検査項目を表したアクティビティ図を**図 2** に示す．われわれは，先行研究で提案されている，モデルベースドテストを現在のテストプロセスに適用し，検査対象を表現した状態遷移機械からテストケースを自動生成する．また，先行研究の不十分な点や相異点を考察したうえで，それらの点について解決方法を示す．本研究と先行研究の相異点として，以下の 4 点を挙げる．

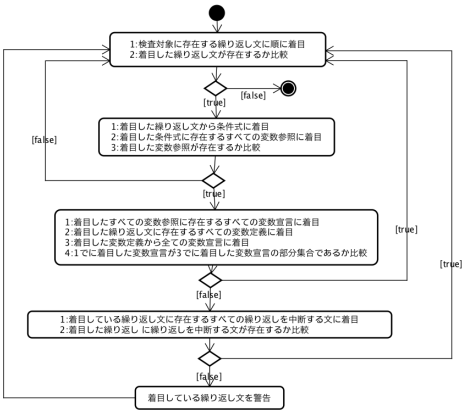

図 2　無限ループ検出機能のアクティビティ図

- 入力とするモデル
- モデルの辿り方の定義
- イベントと生成するソースコード片の対応付け
- 自動生成系の設計

本研究では，検査対象を表現したモデルを基に，モデルの辿り方と，生成すべきコードを算出する．テストケース自動生成の流れを以下の**図 3** に示す．

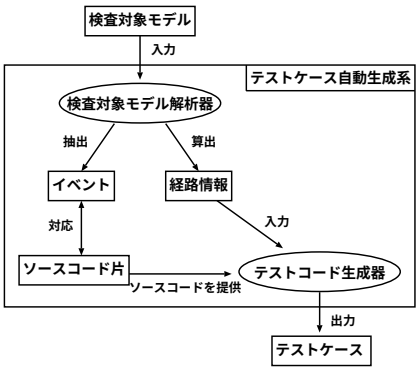

図 3　テストケース自動生成系の流れ

われわれの研究では，**図 3** の流れで，多数のテストケースを自動生成し，検査コードの正当性を確認することである．現在テストケース自動生成の入力となる検査対象モデルから，検査コードを自動生成する方法が提案されている．しかし，同じ検査対象モデルからテストケースと検査コードを自動生成した場合，検査コードの正当性を単純に確認することはできない．検査コードの正当性を確認するには，検査対象モデルが妥当であるかを確認する必要がある．検査対象モデルの妥当性の確認は以下の観点で行なう．

- 検査対象モデルが正しい方法で記述されているか
- 検査項目に対する仕様に誤り，漏れがないか

検査対象モデルが正しい方法で記述されているかの確認は，状態遷移機械の状態が正しい Java の構文要素の順で出現するかなどで判断をする．また，検査項目に対する仕様に誤りや漏れがないかの確認は，検査対象モデルからテストケースを生成し，生成したテストケースが仕様を満たしているかで検査項目に対する仕様に誤り，漏れがないかを確認する．仕様を正しく表現できていなかった場合，検査対象モデルを作成し直す必要がある．検査対象モデルの妥当性が確認された場合，検査コードの正当制を確認する．以上で説明した，テストケースや検査コードの関係性を**図 4** に示す．

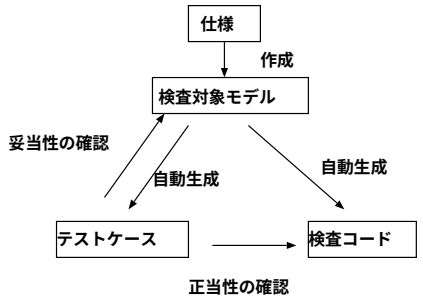

図 4　テストケース使用目的

5　モデルベースドテストの実現

5.1　検査対象を表したモデル

先行研究では，検査処理を表現するモデルとしてアクティビティ図を使用していた．入力がアクティビティ図である先行研究と，入力が状態遷移機械である本研究の相異点を次に示す．

- 処理の記述場所
- 経路の辿り方

アクティビティ図では，頂点に処理が記述されていた．しかし，今回われわれが使用する状態遷移機械では，辺に次に出現すべき構文要素がラベル付けされ記述されている．このことから，状態遷移機械を辿る経路ごとに，プログラムとして正しい構文要素の順番で処理の情報を得ることができる．

アクティビティ図では経路を辿る際，終端である頂点は，経路の最後であることを意味する．しかし，状態遷移機械では，スタックを利用し入れ子を表現しているので，終端である頂点が必ずしも経路の最後であるとはいえない．このことから，経路の最後となる頂点へ移動する際には，「スタックが空である」という条件が必要である．

5.2　自動生成の流れ

テストケース自動生成系の入力となる状態遷移機械からテストケースを自動生成する．自動生成は，状態遷移機械の経路を列挙し，その後全ての経路に対して，入力の情報を基にコード片を組み合わせることで行なう．

状態遷移機械の経路を列挙する際，無限に経路を辿ることのないようスタックに格納する回数に制限が必要になる．

5.3　モデルの辿り方

先行研究では，モデルの経路はアクティビティ図の条件判定にパスを割り振り，そのパスの組合せにより算出すると考察がされていた．しかし，パスの組合せの情報をどのように取得するかについては考察がされていなかった．本研究では，前状態・後状態をスタックに格納する．また，スタックに格納された情報から，状態遷移機械の開始から終了までの経路を取得する．例として，**図 1** の「if 文の条件式内に後置式が存在する場合警告する」検査項目を生成系に入力した際の処理を考える．前状態・後状態の情報を取得し，リストに格納する．格納したものを**図 5** に示す．

開始 ==> Start
Start ==> BeforeExpression
Start ==> **終了**
BeforeExpression ==> InfixExpr
InfixExpr ==> InfixExpr
InfixExpr ==> Stackpop
InfixExpr ==> warning
Stackpop ==> InfixExpr
Stackpop ==> Start
warning ==> Stackpop

図 5　前状態と後状態の情報

開始状態から終了状態までを通過する経路を算出する．**Stackpop** 状態から他の状態へ遷移する際はスタックに格納されている情報に応じて，どの状態へ遷移するかを決定する．また，スタックに情報が格納されている場合，遷移する際に格納されている情報を pop する．

結果，**図 6** のように経路が算出される．

開始 ==> Start ==> BeforeExpression ==> InfixExpr
==> Stackpop ==> Start ==> **終了**

開始 ==> Start ==> BeforeExpression ==> InfixExpr
==> warning ==> Stackpop ==> Start ==> **終了**

開始 ==> Start ==> BeforeExpression ==> InfixExpr ==> InfixExpr
==> Stackpop ==> InfixExpr ==> Stackpop ==> Start ==> **終了**

・
・
・

図 6　経路の情報

図 6 のような経路を算出した際に，各経路情報に対して Warning 状態を通過しているかを確認する．経路に Warning 状態が含まれていた場合は，警告すべきテストケースを生成する必要がある．また，経路に Warning 状態が含まれていない場合は，警告すべきでないテストケースを生成する必要がある．

5.4 作成方法

5.3 節で導き出した全ての経路に対してテストケースを作成する．テストケースの生成に，状態遷移機械のイベントを利用する．状態遷移機械のイベントは構文要素などの特定の意味を表現した要素なので，特定のソースコード片と対応づける事ができ，状態遷移機械の開始から終了までのイベントに対応づけたソースコード片を組み合わせることでテストケースを作成する事ができる．

例として，**図 1** の状態遷移機械で **Start** 状態，**BeforeExpression** 状態，**InfixExpr** 状態，**warning** 状態と遷移する場合，**IfStatement**，**Infixexpression**，**Postfixepression** のイベントを通過する．このイベントのそれぞれに if 文，式, 後置式のソースコードを対応づけておくことにより，上記のイベントの特性を含んだテストケースを生成することができる．

テストケース例：条件式に後置式が存在

```
public static void main(String[] args) {
  int i = 0;
  if(i++ == 0);
 }
}
```

5.5 テストケース自動生成系

テストケース生成系を作成するには，与えられたイベントに対応するソースコード片を用意する必要がある．イベントに対してソースコード片を定義する際に必要な情報として以下の 2 点が挙げられる．

- 初期定義の情報
- 可変部分と不変部分の情報

イベントには，始めに生成すべき初期定義と，初期定義に対して可変部文と不変部分の情報を定義しなければならない．**図 7** にイベントとソースコード片の対応例を示す．

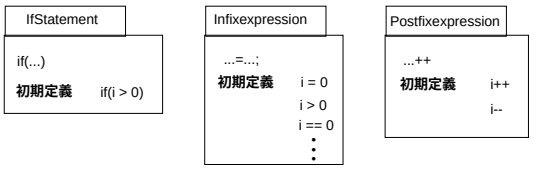

図 7　イベントとソースコード片の対応例

図 7 では，「…」が可変部分を表現しており，他の記述が不変部分を表現している．

6 考察

6.1 新しい検査項目に対する考察

テストケース生成には，イベントに生成するソースコードを定義する必要がある．現在定義がされているイベントだけで表現ができる検査対象であればテストケースを生成することが可能である．しかし，未定義であるイベントを記述しなけば表現ができない検査対象に対してはテストケースを生成することができない．未定義であるイベントに対してソースコード片を随時追加していく必要がある．

6.2 適応後のテストプロセスに対する考察

現在のテストケース生成では，「仕様書からテストケース作成」の段階において，仕様書を基に構文要素の種類，制御の流れ，データの流れの組合せなどを考慮し，多数のテストケースを全て手作業で生成を行なわなければならない．これらは手作業で行なっており，テスト担当者の労力が大きい．それに対し，本稿で提案したテストケース生成方法では，仕様書を基に作成した状態遷移機械を生成系に入力することで，多数のテストケースを生成することができる．以上のことから，現在のテストプロセスに対して少ない労力でテストを行なうことができる．

また，テスト対象を抽象化したモデルである状態遷移機械を基にテストを生成するので，手作業で生成する現在のテストプロセスと比べて，テストカバレッジを高めることができる．

7 おわりに

本研究では，JCI の検査項目に対するテストの支援を目的として，テストケースを状態遷移機械から自動生成する方法を提案した．状態遷移機械内の経路の情報を導き出す方法を確立し，状態遷移機械のイベントにソースコードを定義することでテストケースを生成する手法を提案した．

今後の課題は，テストケース自動生成系を実現することである．また，実際にテストケース自動生成系を利用し，提案した自動生成方法が妥当であったかを考察する必要がある．

参考文献

[1] I. K. El-Far, J. A. Whittaker, *Model-based softwaretesting, In Encyclopedia on Software Engineering(edited by J.J. Marciniak)*, Wiley, 2001.

[2] 浦野彰彦, 沢田篤史, 野呂昌満, 蜂巣吉成, 張漢明, 吉田敦 “デザインパターンを用いたソースコードインスペクションツールのソフトウェアアーキテクチャ設計,” ソフトウェア工学の基礎 *XVII*(日本ソフトウェア科学会 *FOSE2010*), pp. 15-24, 2010.

[3] 二宮剛史, “Java ソースコードの CDI(Code Inspection) ツールの開発 －コードインスペクションツールに対するテストプロセスの改善－,” 南山大学大学院数理情報研究科 2009 年度 修士論文 (OJL 成果報告書), 2010.

[4] 山本陽司, “Java を対象としたソースコードインスペクションツールの開発 －テストデータ自動生成ツールの開発－,” 南山大学大学院数理情報研究科 2010 年度 修士論文 (OJL 成果報告書), 2011.