自動でテストパターンを生成する単体テスト環境の構想

松本真樹† 大野和彦†
佐々木敬泰† 近藤利夫†

1. はじめに

金融工学などで用いられるソフトウェアは出来る限り短期間で開発する必要がある．しかし，近年そのようなソフトウェアはより精細な計算が求められており，開発の大規模化が進んでいる．それに伴い関数レベルのモジュール数も増大しており，各関数の単体テストの人的・時間的コストも大きくなっている．単体テストは関数レベルでホワイトボックステストを行うためシステムのバグを早期に発見できる効果があり，潜在的なバグによって開発期間が長期化してしまう可能性を減らすことができる．一方，単体テストの作業自体は比較的単調であり，自動化する試みがある．しかし，それらはソースコードの情報だけを使いテストパターンを生成しているため，単純なテストしか行うことができず完全な自動化は難しい．そこで本稿では，ソースコードから生成したテストパターンを仕様書を使い補正する自動単体テスト環境を提案する．

2. 背景

2.1 単体テスト

ソフトウェア開発ではウォーターフォール・モデルやスパイラル・モデルなど様々な開発モデルが用いられている．しかし，どの開発モデルでも最下層の開発は関数レベルで行われる．具体的には，プログラマは仕様書に従って関数を作成する．そして，その関数が仕様書に従って正常に動作するかを単体テストによって確認する．単体テストでは，関数の構造を解析し各命令が意図した通りに動くか確認するホワイトボックステストが用いられることが多い．

単体テストの実施者は最初にテストパターンを作成する．これは入力値と，その入力値で正常に動作した場合の出力値の組み合わせである．具体的には関数の引数やグローバル変数，データベースやファイルの値などが入力値や出力値となる．テスト実施者は一個の関数に対して，コードの構造や仕様などから複数のテストパターンを作成する．そして，各テストパターンの入力値を用意して関数を実行し，動作後の出力値がテストパターンと一致するか確認する．近年は開発期間は短くなっており，より少ないテストパターンで単体テストを行うことが求められている．従って，数が少なくても効率的にテストを行うテストパターンを作成する必要がある．しかし手動で効率的なテストパターンを作成することは難しく，またテストの漏れが発生してしまう恐れがある．

```
1.int func(int arg1, int arg2) {
2.  int rtn = 0;
3.  if (arg1 > 0)
4.    rtn += arg1;
5.  if (arg2 < 0)
6.    rtn -= arg2;
7.  return rtn;
8.}
```

図 1 サンプルコード

2.2 ホワイトボックステスト

ホワイトボックステストとは関数内の構造を把握し，各命令を最低一回は実行し検証するテストである．命令の網羅の仕方には命令網羅や条件網羅，経路網羅がある．経路網羅は他の命令網羅より網羅率が高く，全ての経路を最低一回は実行する必要がある．例えば**図 1** の 3 行目や 5 行目の条件式で，3 行目が真，5 行目が偽となるテスト，3 行目が偽，5 行目が真となるテスト，3 行目が真，5 行目が偽となるテスト，3 行目が偽，5 行目が真となるテストの 4 回を行う必要がある．条件網羅は理想のテストであるが，テストパターンの数は条件分岐の数に対して指数関数的に増加し，手動で行うことは現実的に難しい．また Jlint[1] 等の自動化ツールもあるが，デッドロックや継承のチェックなどのコードだけで判断できる項目に対してしかテストを行えず，3 行目や 5 行目の分岐式の正当性を検証することは不可能である．従って，手動によるテストパターンを作成することが必要になる．

† 三重大学大学院工学研究科情報工学専攻

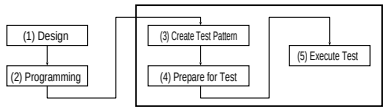

図 2 関数実装における作業フロー

3. 単体テスト自動化環境

3.1 概　　要

2 章で述べたように手動で経路網羅を行うことは難しい．そこで我々は経路網羅を行うようなテストパターンを自動的に生成し実行する環境を提案する．本環境を用いることで手動による作業を極力減らしソフトウェア開発の効率化をはかれ，テストの漏れも少なくなる．しかし，コードから自動でテストパターンを生成するだけでは入力値に対して正常な出力値を決定することが不可能である．そこで本環境では出力値を決定する方法として仕様書の情報を用いる．従って本環境では，(1) コードから経路網羅を行うテストパターンを生成し入力値を決定し (2) 仕様書から生成したテストパターンの出力値を決定し (3) 完成したテストパターンの実行を行う．本環境では (1) を**ソースローダ**，(2) を**仕様書ローダ**，(3) を**テスト実行**とする．

通常の作業工程と本環境を用いた場合の違いを**図 2** に示す．(1) は関数仕様書の作成であり，(2) はプログラムの作成になる．(3) はテストパターンの生成で，(4) は関数外の変数に入力値を設定する等の準備，(5) はテストパターンを実行し動作の確認を表す．通常は手動で全ての行程を行う必要があり，また (3)-(5) はテストパターン数だけ繰り返し行う．一方で本環境を用いた場合，黒枠線内の (3)-(5) は自動で行われ開発の効率化を図れる．

3.2 ソースローダ

ソースローダでは関数のプログラム構造を解析しテストパターンを生成する．ソースローダは最初に全ての命令経路を探索し，各命令経路において実行される分岐命令を抽出する．そして各分岐命令において，判別式で用いられている関数や変数，その後の命令経路を調べる．そして判別式で用いられる変数がどの関数外の変数 (関数の引数やグローバル変数) に依存しているかを解析し，期待する命令経路を実行するように値を決定させる．ソースローダはこの値を入力値としてテストパターンを生成する．ここで生成されたテストパターンは出力値が決まっていないが，これは仕様書に依存するので仕様書ローダで定める．

3.3 仕様書ローダ

ソースローダで作成したテストパターンは，入力値の値は決定しているが出力値の値は決定されていない．そこで仕様書ローダでは仕様書を読み込み出力値を決定させる．読み込む手段として (1) 仕様書を言語解析し自動的に出力値を得る (2) 仕様書を元にツールから手動で入力してもらう (3) ユーザにはツール用いて仕様書を作成してもらい，そのツールが仕様書を出力すると同時に必要な出力値を決定する，が考えられる．(1) は仕様書の中から出力値を自動で抽出するので，仕様書の言語解析を完全に行う必要があり現実的でない．(2) については，経路網羅を行うため非常に多くのテストパターンが生成され，ユーザに対する負担が大きい．(3) については，ユーザが仕様書をツールで作れば出力値が自動で決定されることになる．従ってユーザの負担も (2) より少なくなる．しかし仕様書にはある程度規則性がありツールで作成できるが，記述に対する自由度が下がる恐れがある．従って，フリースペースのように自由に記述できる仕組みが必要である．

本環境では (3) をベースに (2) で補完する方法を採用する．(1) と同じ理由で，自由に記述した内容から出力値を決定させることは難しいので，(2) でユーザに出力値を補完してもらう．最終的には (3) のみで出力値を得ることを目指す．

3.4 テスト実行

JUnit[2] など生成したテストパターンから単体テストを自動で実行する環境は存在する．従って本環境では生成したテストパターンを変換し，JUnit などを使いテスト実行を実際に行う．ただしどのソフトウェアで単体テストを実行するかについては調査が必要であり今後の課題である．

4. おわりに

本稿では，自動で単体テストを行う環境について提案した．従来はプログラムコードのみでテストパターンを生成していたため比較的単純なテストしか行えなかったが，本環境では仕様書も含めてテストパターンを生成することを想定している．従って，より現実的なテストを行うことができる．今後はソースローダや仕様書ローダ，テスト実行の詳細な設計を行い実装していく必要がある．また，ファイルやデータベースなども入出力値としてテストパターンに加える手法の開発も行っていく必要がある．

参 考 文 献

1) http://jlint.sourceforge.net/: Jlint - Find Bugs in Java Programs.
2) http://www.junit.org/: JUnit.