Software Fault Injectionを用いた開発時テスト支援環境

Software Testing Support Environment using Software Fault Injection

黒田滋樹 † 柴山悦哉 †

Masuki KURODA Etsuya SHIBAYAMA

† 東京工業大学大学院 情報理工学研究科 数理・計算科学専攻

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

{kuroda2, etsuya}@is.titech.ac.jp

ネットワークなどの外部デバイスを使用するプログラムでは，常に物理的な異常が発生する可能性があり，異常時に備えた復旧などの例外処理が定義される．しかし物理的な異常時の処理は非同期に発生し再現性に欠けるため，開発時におけるユニットテストが難しい．本研究では外部デバイス異常の例外処理のテストをテストケースとして記述可能とした．開発者はテストケース中で異常の種類，再現性を確保するため異常発生のタイミングを指定することで，指定した外部デバイス異常時のテストケースを，既存の正常時のテストケースと同様にテスティングフレームワークである JUnit 上で実行可能とした．

1 はじめに

近年のプログラミングスタイルとして，コーディングと並行してユニットテストを行う手法が一般的になりつつある．この手法ではテストのモジュール化や効率化のためテストケースを用いることが多く，テストケースを機械的に実行する代表的なテスティングフレームワークとして XUnit がある．本論文で単にテストケースと記述した場合，XUnit で用いられるテストケース，すなわち仕様から得られた特定の状況におけるプログラムの挙動を確認するプログラムを指している．

テストケースには実行によって他のテストケースの状態に影響を与えない独立性と，もう一度同じテストを行った際に同一の結果が得られる再現性が求められる．そしてテストケースではプログラムが正常な動作をしている際の仕様だけでなく，異常が発生した際の仕様も定義される．例えば不適切なファイル名が指定された場合にはある例外を投げる，という仕様であれば，異常なファイル名を与えて発生する例外を確認するテストケースを記述，実行すればよい．

しかし，XUnit ではネットワーク切断のようなプログラム外部の異常が発生した際の処理をテストすることは容易ではない．ハードディスクやネットワークといったプログラムの外部デバイスに異常を発生させるには，テストケース実行の際に外部に対して異常を導入する必要があるからである．例えばネットワーク経由でデータのやりとりを行っている際にネットワークが切断された場合は，異常を検知して整合性のとれなくなったファイルを削除するという仕様を考える．この場合，特定のタイミングでネットワークが切れる場合という再現性を持ったテストケースは記述できない．

本論文では外部デバイス異常時の処理のテストをテストケースとして記述可能とすることで，XUnit 上で異常時のテストを正常時のテストと同様に行うことを目標とする．今回提案するシステムでは，外部デバイスへのアクセスにプロセス単位で仮想的な異常を導入することで異常時のテストを実現する．そしてテストケース中では，導入する異常の種類と異常発生のタイミングを指定することで，再現性を持ったテスト実行を可能とした．また，ここでは異常の発生するタイミングを指定するため，ホワイトボックステストを仮定している．

2 Software Fault Injection

ソフトウェアの動作を検証するには，正常時の動作だけでなく，異常時の動作も検証する必要がある．Software Fault Injection (SFI) はソフトウェア上でプログラムに故意に異常を導入し，得られる異常時の動作からバグを見積もる手法である．SFI はその手法から二つに大別できる．

2.1 ソースコードレベルの SFI

この手法では，プログラム自体を書き換えることで正常な処理とは異なる（例えば関数の返り値を変更する，実行時に例外を発生させる）処理を埋め込み，異常を導入する．この手法の長所は，異常を発生させたソースコード上の位置とその時のテスト結果の対応がつくことである．これは発生させた異常の位置とテスト結果からソースコード上での修正が必要な点を探す範囲を絞ることができるため，バグの特定の助けとなる．

一方，この手法では変数の自由な改変など導入できる異常の自由度が高い反面，一貫性を保った形で異常を導入することは容易ではない．例えばネットワークが切断された際の挙動を検証するためには，テスト対象プログラムの通信を行う可能性のある全ての箇所に異常を導入する必要があり，もし異常が導入されない箇所があれば一貫性が崩れてしまう．すべての箇所に導入していてはプログラムの規模が大きくなった際に指定するコストが大きくなる．

2.2 低レベルな SFI

これはソースコードなどの情報を用いず，システムコールや基本的なライブラリを改変することで異常を導入する手法である．この手法の長所はプログラムの構造に依存せず異常を導入することができ，異常の一貫性を保ちやすいことである．低レベルな機能はプログラムに依存せず一定であり，その数も限定されるため，すべてに異常を導入すれば異常の一貫性を保つことができる．

低レベルな異常の導入は，導入する対象によっていくつかの方法に分類される．

- **基本的なライブラリの改変** libc などの基本的なライブラリを改変する
- **システムコールの改変** プロセスの実行するシステムコールをフックし，改変する
- **カーネルの改変** カーネルを改変する

本論文で対象としている外部デバイスへのアクセスは基本的にシステムコールを経由してカーネルで処理される．外部デバイスの異常を実現するにあたってのそれぞれの手法の特徴を表 1 に示す．

ライブラリの改変では，ライブラリを経由しないアクセスに対しては異常を導入することができない．また，システムコールの改変ではシステムコールフックによるオーバーヘッドが大きくなる．カーネルの改変では異常の導入に管理者権限が必要となる．

これらの特徴に加えて，共通の短所としてソースコードとの対応を付けにくいため，ホワイトボックステストの実行が困難となる．低レベルの SFI では異常を発生する条件としてメモリの状態や手動によるタイミングの指定が用いられるが，これらを用いて仕様に反した結果が得られても，ソースコード上でどの位置の例外処理を修正すべきかの特定が難しい．

3 設計

第 1 章で述べた XUnit 上で異常時のテストと正常時のテストを同様に扱うため，開発者は以下の手順でテストを行う．ここでは特にテストケースは JUnit を対象とした記述方法を用い，行の指定が可能なインタフェースとして統合開発環境である Eclipse を仮定する．

1. Eclipse の画面上で行を指定することで異常の発生するタイミングを決める
2. 指定したタイミングで発生する異常の種類をテストケース中に記述
3. 異常時のテストケースを JUnit と同様の記法で記述し，通常通りテスト実行

これを実現するため，本システムでは SFI を用いて外部デバイスの異常を導入する．実行されるテストケースは第 1 章で示したテストケースの独立性，再現性の中でも発生する異常の独立性，再現性を満たす必要がある．そして，導入する異常は第 2.1 節における一貫性も保つ必要がある．

本論文ではソースコードレベルの SFI と低レベルの SFI を組み合わせることで前の再現性と一貫性を満たすテスト環境を提案する．ソースコードレベルではテストケースに指定を組み込むことで，異常の種類と発生するタイミングの指定のみを行い，実際の異常導入は低レベルな SFI を実現する別のプロセス（監視プロセス）が行う．二つのプロセスはテスト実行時に通信を行い，テストケースを実行するプロセスから監視プロセスに対して導入すべき異常の指示を送信する．

テストケース上での指定はテスト対象言語への依存が大きく，監視プロセスはプラットフォームへの依存が大きい．この設計により片方の環境が変化した際に必要な変更を抑えることができる．

表 1: 外部デバイス異常における低レベルな SFI の特長

	任意のプログラム	実行速度	管理者権限不要
ライブラリ	△	○	○
システムコール	○	×	○
カーネル	○	○	×

本章では，テストケース上で指定を行うモジュールと，監視プロセスによって実際に異常を導入するモジュールについて述べる．テストケースの独立性については第 3.1 節で，JUnit と同様のテストケース記述方法は第 4 章にて述べる．

3.1 異常の指定

このモジュールでは開発者にソースコードレベルでの異常を指定する手段を提供する．本論文で対象としている外部デバイスの異常を指定するためには開発者はテストケース中で次の二つの条件を指定する．

- **異常の種類**
  異常の発生するデバイス，発生する異常のエラーコードの指定．例えば接続を遮断する IP の指定や到達不能エラー，タイムアウトなど．
- **異常の発生するタイミング**
  テストの再現性を確保するため，異常の発生するタイミングの指定．

これらの指定に加えて第 1 章で述べた独立性の中でも異常の独立性を保つため，テストケースごとに独立した異常の指定手段を提供する．本システムでは異常の指定の有効化，無効化を指定可能とすることでこれを提供した．開発者は異常時のテストケースを記述した範囲でのみ有効化を指定することで，テストケースごとに独立した異常の指定を行う．

指定されたこれらの異常に関する情報はテスト実行前にテストケースのプログラム中に挿入され，実行時に監視プロセスに対して送信される．

3.2 異常の導入

このモジュールでは第 3.1 節の指示を受けて，プロセスに低レベルな異常を導入する．本システムではテストという目的からパフォーマンスは重視しない．そして任意のプログラムに対して適用可能であること，適用に管理者権限が必要ないことから，システムコールの改変を用いて異常を導入する．

システムコールの改変をユーザプロセスで実現するため，実行時にはテスト対象のプロセスの発行するシステムコールをフックする監視プロセスを起動する．これによりテスト対象プロセスのシステムコールの監視，改変が可能となる．(図 1)

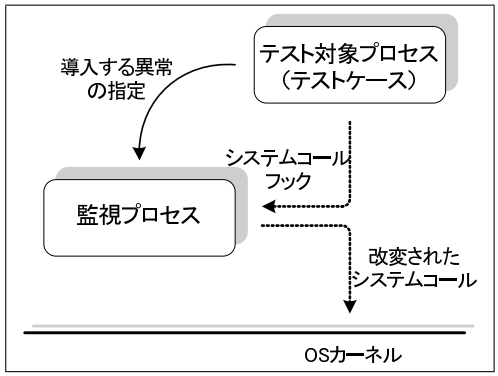

図 1: テストケースから発行されたシステムコールが，異常の指定のとおりに改変される

4 実装

第 3 章で挙げたテストケース記述方法から Eclipse 上の行指定のインタフェース以外で，異常時のテストケース記述と既存のテストケースが同様の書式となる必要がある．本システムでは異常の指定のみをテストケースから記述し，異常の導入はテストケースで記述せず，テスト実行と結果の検証は JUnit の機能をそのまま利用する．つまり，異常を指定するモジュールのみ JUnit のテストケース記述方法に従えばよい．本章ではこの記述方法について述べ，この記述方法によって通信を行う監視プロセスの構成についても述べる．

今回の実装ではプラットフォームは Linux 2.4.27 上として，異常の導入モジュールは strace を拡張，異常の指定モジュールは Java を対象として，JDK 1.5.0 で実装した．

4.1 ラベルの導入

テストケースの作成にあたって，開発者は異常を発生させるタイミングをソースコード上の行にラベルを付けることで指定する．ラベルはソースコード上の行と対応付けられ，各ラベルはその有効無効性とそこで発生すべき異常の情報を保持し，これらは外部のファイルに保存される．

ラベルはテストケース中で抽象化された LabelSet クラスとして提供され，ラベルの情報の取得や変更はこのクラスから行うことができる．本システムはテスト実行前にラベルの行に以下の命令を挿入することで，ラベルが有効な場合にのみ監視プロセスに対して異常の指示を送信する指定を実現した．

```
if(LabelSet.isEnabled("LabelName")) {
  // 監視プロセスに対してラベルに格納された
  // 異常の情報を送信する．
}
```

今回，ラベルのインタフェース指定は統合開発環境 Eclipse-plugin として実装し，画面上ブレークポイントと同様な指定を可能にした．

4.2 テストケースの記述

テストケース中では定義したラベルに対して，ラベルの有効化と異常の指定を行う．ここではテストケースの記述例として，ラベル **L1** にネットワーク切断を定義した場合のテストケースを示す（図 2）．開発者はあらかじめ Eclipse 上で someNetworkAccess() 中にラベルを定義しておく．そしてラベルに指定 IP のホストとの通信が切断される（”1.1.1.1”, ”cut”）設定を行い，有効化する（LabelSet.enable()）ことで，ネットワーク異常時のテストケースが定義できる．また，testCaseCorrect() のようにラベルを有効化しないことで異常が導入されないテストケースを定義できるため，本システムを考慮していない既存のテストケースも変更することなく定義することができる．

テストケース記述後は実行前に第 4.1 節の命令が挿入され，テストが実行されるとテストケース中でラベルの有効化などが行われる．その後，挿入された命令が実行されると，ラベルが有効化されていれば監視プロセスに異常の情報が送信される．

4.3 監視プロセスの実行

監視プロセスはテスト対象プロセスとともに実行することで，対象プロセスの発行するシステムコールをフックし監視，改変を行う．今回システムコールのフックには ptrace を用いた．ptrace は主にデバッグ用途の命令で，指定したプロセスをシステムコール発行の時点で停止させ，発行されたシステムコールの内容を参照することができる．

```
public void testCaseError() {
  LabelSet.setInfo("L1", "host", "1.1.1.1");
  LabelSet.setInto("L1", "type","cut");
  LabelSet.enable("L1");

  someNetworkAccess();

  assertNetworkError();
  LabaleSet.disable("L1");
}

public void testCaseCorrect() {
  someNetworkAccess();
  assertNetworkCorrectAccesses();
}
```

図 2: ネットワーク異常を導入した場合と，正常な場合のテストケース

開発者が異常導入の指示に用いる情報は，ある IP のホストとの通信を切断する，などである．しかし，システムコールはディスクリプタ番号を用いて接続先を指定するため，IP アドレスとの対応がつかず，単一のシステムコールからでは改変すべきシステムコールであるか特定できない．そこで監視プロセスはフックしたシステムコールからディスクリプタテーブルのエミュレートを行う．これによってディスクリプタ番号と IP アドレスやフォルダの対応が得られる．

監視プロセスは常にシステムコールを監視してディスクリプタテーブルの更新を行い，テストケースからの指示を待つ．指示を受信するとそれ以降はシステムコールが異常の導入対象となっていないかマッチングを行う．異常の指示にマッチした場合は，指示どおりにシステムコールを改変する．

4.4 指示の受信とシステムコールの改変

今回，監視プロセスへの指示としてネットワークアクセスの遮断を実装した．監視プロセスに対して送信される指示は，ラベル，異常の種類，接続先 IP アドレスからなる．ラベルの情報は同じラベルからの重複した指示の防止と，特定のラベルで定義された異常の指定を無効化する際に用いる．

監視プロセスはこれらの異常の指示を受信すると，無効化の指示があるまで該当するシステムコールにエラーを導入し続ける．例えば上記のアクセス不可を指定した場合には，対象 IP のディスクリプタに対する connect, send, recv などのシステムコールを失敗させることでアクセスの遮断を実現する．

5 関連研究

SFI を用いた研究のうち，低レベルな SFI を実現するための手段として Xception [1] が提案されている．Xception はカーネルを変更することで，プロセッサの動作に異常を導入している．例えば，浮動小数点ユニットやメモリ管理ユニット，データバスなどに一定の確率でビットフリップの異常導入を可能としている．この異常導入を数万回行い，結果の誤りやハングアップの発生割合を得ることで，プログラムの堅牢性を見積もることができる．

しかし，これらの検証手法はブラックボックス化した検証手法を提案している．そのため，今回対象としている開発としているテスト結果に応じてソースコードを修正しながら開発を進めるスタイルには適さない．

ソースコードレベルの SFI を実現する手法として FIRE [2] が提案されている．FIRE は OpenC++のトラップ機能を用いてソースコード上の変数や返り値の変更，例外発生を用いて異常を導入する．例えば関数の呼び出し時に返り値を変更する命令を実行することで，返り値を自由に書き換えることができる．

この手法は外部デバイスの異常に限らず自由度の高い異常導入が可能となる．しかしその分，実際に発生しうる一貫性の取れた異常を定義することは容易ではない．

SFI は MOCK[3][4] を用いることでも実現可能である．MOCK とはプログラムのインタフェースは同じで内部の実装が異なる張りぼてであり，テストの時にはテスト用の MOCK にすり替えることで異常を導入できる．この手法はプログラムの設計段階で MOCK を導入しやすいような設計を行った場合には，拡張性に優れた手段である．しかしその種の設計なしに作られたプログラムでは，ソースコードレベルの SFI と同様に一貫性を保ちにくい．そもそも MOCK は SFI のための技術ではないため，本システムの低レベルな異常導入との併用が可能である．

6 まとめと将来課題

本論文では外部デバイスの異常を対象として，異常が発生した際の挙動をテストする環境を提案した．テストは Eclipse-plugin を用いて，JUnit のテストケースとして定義可能とし，既存の正常時のテストケースと同時に記述できる．異常のタイミングをソースコードの行で指定することでテストの再現性を確保し，システムコールに異常を導入することで異常の一貫性を実現した．

今回提案したテストケースの定義方法では，タイミングの指定にソースコード上の行を用いた．しかし，外部デバイスの異常は任意のタイミングで発生しうるため，それらをテストするには多くの行を指定しなければならず，ラベルごとのテストケース記述も負担となる．そこで行指定の負担を軽減するための手法としてテストケースの自動生成が挙げられる．例えば外部デバイスへアクセスしうる箇所を抽出し，異常発生のタイミングをその前後に指定したテストケースの雛形を自動生成することで記述負荷が軽減できる．

また，異常発生の条件として，時間的な指定だけでなくプロトコル単位の指定への拡張が考えられる．ネットワークを利用するプログラムのテストでは，単純な切断だけでなくプロトコルのどのタイミングで異常が発生するのかも指定可能としたほうがテストの記述力が増す．

参考文献

[1] Carreira, J. and Madeira, H. and Silva, J.G. : Xception: a technique for the experimental evaluation of dependability in modern computers Software Engineering, IEEE Transactions on Volume: 24 Issue: 2 pp125-136 1998

[2] Martins, M. and Rosa, A.C.A. : A fault injection approach based on reflective programming in Proceedings of the Dependable Systems and Networks, pp:407-416 2000

[3] David Saff and Michael D. Ernst : Mock object creation for test factoring in Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering pp:49-51 2004

[4] Mackinnon, T. and Freeman, S. and Craig, P. : EndoTesting: Unit Testing with Mock Objects eXtreme Programming and Flexible Processes in Software Engineering - XP2000