動的解析を利用した正常系解析と正常系表示エディタの開発

中山 心太
NTT 情報流通プラットフォーム研究所
E-mail: nakayama.shinta@lab.ntt.co.jp

Dynamic code path analysis and its visualization for normal code path.

Shinta Nakayama
NTT Information Sharing Platform Laboratories
E-mail: nakayama.shinta@lab.ntt.co.jp

概要 IT の現場で運用されているプログラムは、正常系が 2 割、異常系 8 割程度の比率で記述されている。C 言語などの既存言語ではプログラムフローの制御と、エラートラップが同一の記法(if 文)で記述されており、静的解析においては区別がつかない。そのため、新たにプロジェクトに配属された人にとっては、運用されているコードを読んでも、正常系も異常系も同様に見えてしまい、プログラムの動作を把握するのが難しい。そこで、テストコードは正常系と異常系が仕様書から定義できるため、テストコードを用いてプログラムの動的解析を行うことによって正常系のみを効率的に抽出し、正常系のみを表示するエディタを提案する。これにより、短期間にプログラムの動作を把握することができ、プロジェクトの開発効率の向上が期待できる。
キーワード 正常系解析、動的解析、コードリーディング

1 背景

プログラムの大規模化が著しい昨今、既存コードの流用やフレームワークの利用が一般的になり、プログラマが 1 からコードを書くことは稀となっている。たとえば、プログラマが新たに部署に配属された場合、そこには既に開発されているプロダクトが存在し、多くの人の手を経たソースコードが存在している。新人プログラマがまず取り組むべき仕事は、おおよそ以下の流れになっている。

・正常系コードの把握によるプログラムの構造把握
・異常系コードの把握
・デバッグ
・機能追加

自らが 1 から書いたコードであれば正常系の把握は容易だが、他人が書いたコードの正常系把握は非常に難しい。適切な変数名、関数名をつけた上で十分なコメントが付いていれば、人間が確認することで正常系か異常系化の判断は可能であるが、大規模なプログラムにおいて目視で確認を行っていてはプログラムの構造把握だけで数日が経過してしまう。

また製品品質のプログラムにおける異常系は 8 割を占めるといわれ、金融系のミッションクリティカルなシステムでは 9 割 5 分が異常系といわれている。それゆえ単純にソースコードを眺めていては異常系が多すぎて機能把握はままならない。そのため新人プログラマがいち早く開発に加わるために、正常系コードの自動抽出が求められている。そこで本稿では正常系コードを表示する手法を検討し、正常系コードを表示するエディタを開発する。

2 課題

2.1 正常系抽出の課題

現行の主流なプログラミング言語において、言語レベルでは正常系を定義できてない。たとえば、if 文は「制御の分岐」と「例外の分岐」の両方に利用され、両者の区別はソースコードのレベルでは区別が付かない。たとえば、起動オプションの差異による処理の分岐と、異常な入力値を検出するための Validator 的な分岐の区別を静的解析において行うことは難しい。また例外処理などの try catch 構文も、例外を捕捉するのに利用されるが、必ずしも catch 文の中で異常系処理を行うわけではない。catch 文の中で異常系に遷移するためのフラグを立て、別の箇所で異常系処理が行われることがある。このような場合、catch 文の中のみを異常系として判断することはできない。

現代のプログラミング言語は処理の分岐を基本的な考え方として設計されており、正常系と異常系を正確に記述するすべを持たない。そのため静的解析を利用してプログラムの正常系を抽出することは難しい。

2.2 異常系抽出の課題

正常系と同様に現代のプログラミング言語においてもまた言語レベルでは異常系を定義できない。前述のように Validator のような利用の仕方もあれば、Catch 文の中に存在することもある。また、接続先サーバが存在しなかった場合のサーバ復帰待ちなどの場合、ソースファイルが丸ごとひとつ異常系である場合もある。したがって、異常系には様々な粒度が存在する。たとえば、if 文のブランチの中、else 文のブランチの中、関数単位、クラス単位、ファイル単位、モジュール単位など、様々な粒度で異常系が存在する。そのため、静的解析において異常系を抽出することはやはり難しい。

3 先行研究

直接的な研究ではないが、エディタとしてKameTLのデバッガと、フラクタルを利用したコード重要度の可視化について取り上げる。

3.1 実行された行を可視化するデバッガ

従来のブレイクポイントのようなプログラムを停止させるデバッグ手法は、ゲームのようなインタラクティブ性の高いリアルタイムシステムでは、インタラクティブ性を阻害するため利用することができない。

丹野の開発したリアルタイムゲーム記述言語KameTLのデバッガ[1]では、実行中の行に対してハイライトを行うことで、インタラクティブ性を阻害せず実行箇所の可視化が可能である。

しかしKameTLデバッガの可視化手法は、プログラムのデバッグを目的にしており、コードリーディングには用いることができない。しかし実行された行を動的に可視化するという仕組みは有用である。

3.2 フラクタルに基づくコードの重要度可視化

小池のFractalViews[2]は、フラクタルのアルゴリズムを用いて、編集中の行と関連性の低い行を縮退させることにより、ソースコードの可読性を向上させるエディタの研究である。

たとえばC言語のswitch case文では、1つのswitch文と多数のcase文からなる。case文が多数あった場合、switch文とcase文が大きく離れることがある。このような場合、FractalViewsはswitch文と現在編集中のcase文のフォントを大きくし、その他の行のフォントを小さくする。これによりプログラムの構文における関連性と、エディタの表示を合わせることができ、効率的にプログラミングを行うことが可能になる。

しかし、FractalViewsはプログラミングを行うための可視化技術であり、構文レベルでの解析を中心としいる。そのため、正常系を効率的に表示するために用いることはできない。しかしプログラマにとって重要な箇所を強調して表示するという考え方は有用である。

4 アプローチ

正常系とはソースコードが実行され、人間が期待していたとおりの挙動を示すことをいう。このような期待する動作について記述されたものが仕様書である。そのため、正常系の定義は仕様書に基づいて行うことができる。

4.1 テストコードは仕様書から作られる

現代のプログラミング言語では、ソースコードのレイヤで正常系と異常系を区別して記述することはできないが、仕様書のレイヤでは区別して記述することができる。またテストコードは仕様書に基づいて作成されるため、テストコードのレイヤでは正常系や異常系の情報は保持されている。そのため、正常系や異常系のアノテーションの付いたテストコードを用いて、ソースコードの検査を行うことで、ソースコード内の正常系や異常系を特定できる可能性がある。

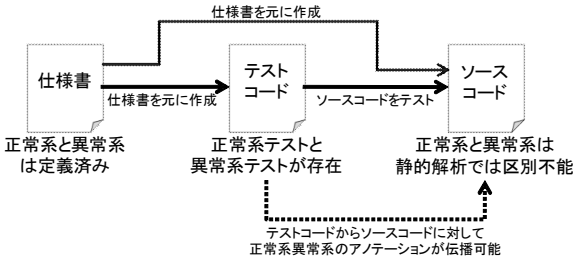

図1:テストを実行することにより、正常系を特定する

4.2 コードカバレッジの利用

テストコードに対して正常系か異常系かのアノテーションを付与する。そして、テストコードを実行する際にコードカバレッジの計測を行う。これにより、テストコードが実行された箇所は正常系または異常系であると定義することができる。以上の流れを図1に示す。

正常系のテストコードを実行されている限りは、異常系が実行されることはまずないため、正常系テストのコードカバレッジを計測することで、実行された箇所を正常系と仮定することができる。また逆に異常系テストを実行下最のコードカバレッジから、ソースコード中の実行された箇所を異常系と仮定することができる。

さらに、2割の箇所が8割実行される経験則[1]から、コードの実行頻度を計測することで、正常系の中でも重要な箇所の抽出を行うことができる。

4.3 エディタへの反映

現在のエディタは静的解析の結果をベースとしたアノテーションが行われており、構文のハイライトなどが行われている。静的解析はインクルードファイルの展開やパーサを利用した型チェックなどが行われており、コンパイルとほぼ同等のことが行われている。

しかし動的解析が行われていないため、前述のようにソースコードに対して正常系、異常系というアノテーションを行うことができていない。また、実行頻度にあわせたハイライトなどもできていない。

一方、今日ではOMake[2]やJenkins[3]などの自動ビルド、自動テストが行えるCI（Continuous Integration 継続的インテグレーション）環境が整ってきた。CI環境とエディタを組み合わせることで、ソースコードを書きながらにして自動でビルドして、自動でテストを行うことができる。テストを行った際にコードカバレッジを計測し、エディタに反映することで、正常系のみを動的にハイライトして表示することができる。

また、プログラムを書き換えて正常系コードのフローが変化したとしても、CI環境によってコードカバレッジを追跡するため、変更に対して追従することができる。

5 実装

実装を図2に示す。エディタにはvimを採用した。CI環境にはOMakeを利用し、-Pオプションを利用しソースコードの改変を監視し、自動ビルドとテスト(main関数の

1 パレートの法則
2 http://omake.metaprl.org/
3 http://jenkins-ci.org/

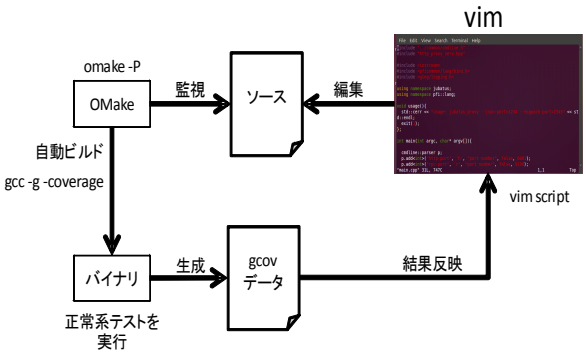

図 2:vim と CI 環境の連携

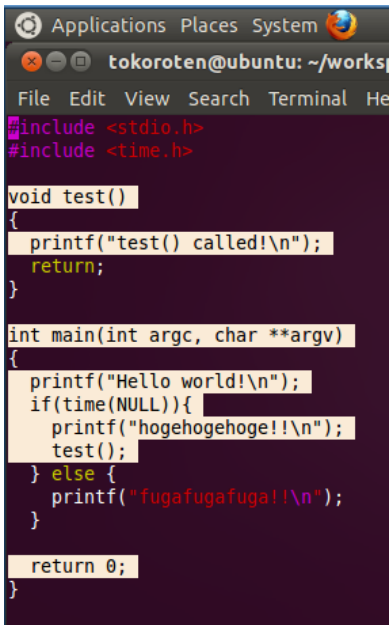

図 3:VimScript によるコードカバレッジの可視化

実行）を行った。コードカバレッジの計測は gcc の coverage オプションと gcov を利用した。gcov が出力したコードカバレッジ情報は VimScript を利用して自動的にエディタに反映される[4]。main 関数を正常系テストであると仮定した際の、コードカバレッジの可視化の結果を図 3 に示す。実行された行が白くハイライトされている。

6 発展

6.1 カバレッジ差分による特徴抽出

カメラを固定して予め背景を撮影しておけば、画像の中に人が現れたときに、背景との差分から人がいる部分だけが抽出できる。同様の処理もコードカバレッジに対しても行うことができる。

6.1.1 集合演算による、上位レイヤの正常系抽出

正常系のコードカバレッジは、変数やクラスなどの初期化処理と正常系自体の処理を含む。逆に異常系のコードカバレッジは、初期化処理は含むが正常系処理は含まず、異常系処理が行われる。

すなわち、コードカバレッジを集合と見立てて、正常系コードカバレッジ∩¬異常系カバレッジという演算を行うことで、正常系コードカバレッジから、変数やクラスの初期化処理を取り除くことができる。演算の様子を図 4 に示す。これにより変数やクラスはブラックボックスとして取り扱うことができ、より高次の正常系処理のみを抽出してコードリーディングを行うことができる。

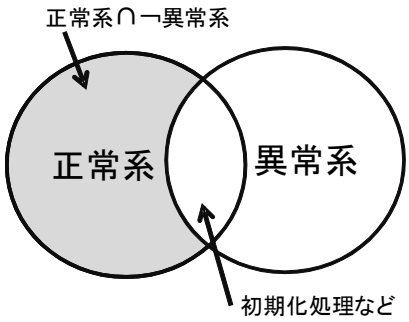

図 4:テストコードカバレッジ同士の集合演算

6.1.2 正常系テスト同士の演算による、機能差分の抽出

前述の手法は、正常系テストと異常系テストの組み合わせだけでなく、正常系同士でも実現可能である。たとえば、起動オプションの差異によって処理が分岐するようなプログラムの場合、起動オプションごとのコードカバレッジの差異を集合演算することによって、起動オプションが影響を及ぼす箇所のみ抽出することができる。すなわち、正常系同士のコードカバレッジの比較により、機能とソースコードのマッピングを行うことができる。

たとえば、ある起動オプションが様々な場所で参照されており、あまりにソースコードの広範囲にわたって影響を及ぼしているようであれば、設計が悪い可能性がある。

6.2 CI 環境からコードカバレッジの流用

現在のコードカバレッジテストは、テストに抜け漏れが無いことを確認するための指標として利用されており、本提案のようなコードにアノテーションをつけるための仕組みとしては利用されていない。利用されていたとしても、テスト漏れが起こっている箇所を探してテストを追加するために利用される。そのためコードカバレッジの統計情報は利用されているが、どのような経路を通ったかというテストごとの生のコードカバレッジの情報は捨てられてしまっている。

近代的な CI 環境や統合開発環境ではコードカバレッジは標準で搭載されているため、本手法が適応でき、捨てられてしまっているコードカバレッジの情報を再利用できる可能性がある。

7 まとめ

商用プログラムでは異常系が八割を占め、正常系を把握することが困難になってきている。また現行のプログラミング言語は、言語のレイヤで正常系や異常系を定義することができない。しかし仕様書の中においては正常系や異常系の定義は可能である。また、テストプログラムは仕様書から生成されるため、正常系のテストプログラムというものは定義可能である。したがって、正常系のテストプログラムのコードカバレッジを調査することで、実行された箇所が正常系であるという仮定をすることができる。

4 発表中に行ったデモは以下で公開している。
http://www.youtube.com/watch?v=5aI3DOwqSQI

そこで、vim と VimScript を用いてコードカバレッジの可視化プログラムを作成し、OMake を用いた CI 環境と組み合わせ、正常系のコードカバレッジを可視化できる仕組みを作成した。

正常系の可視化に関しては、現在 OSS プロダクトを利用して評価中である。

参考文献

[1] 丹野 治門: “ゲームシステム記述言語kameTLにおけるリアルタイムデバッグ機構”, 情報処理学会第 49 回プログラミングシンポジウム報告集, pp.17-24, 2008. 2008

[2] Hideki Koike: “Fractal Views: A Fractal-Based Method for Controlling Information Display”, ACM Trans. on Information Systems, Vol.13.No.3, 1995.

質疑応答

東京大学 川中：腕のいいプログラマが書くと色がきれいになったりして、腕の良さを見分けられるのではないか？

確かに腕の良いプログラマはアスペクト志向などで、異常系を綺麗に切り分けて書くことができる。正常系と異常系が縞模様になっていたら腕が悪い、大きく分かれていたら腕がいいという評価指標にはできるかもしれない。

東京大学 荒川：正常系可視化のエディタをクロスプラットフォームにする予定はあるか？Eclipse を使えばコードカバレッジを出せるのでそれを使えば良かったのではないか？

はじめは Eclipse で開発を行おうとしていたが、Eclipse を常用していない上に、プラグインの作成が複雑だったため、実装の手間を考えて vim と VimScript を採用した。普段は VisualStudio で開発しているので、できればそちらで実装したい。

電通大 岩崎：現在は vim の制約でラインハイライトするだけだが、より自由度がある環境ではどのような表現方法があるか？

実行されてない箇所は隠すように考えていた。例えば、実行回数が多いところは大きくなるとか。異常系は見えないのが一番良いんじゃないか。

電通大 岩崎：そのへん使おうとするエディタによっては大きくできなかったりするのではないか？

Visual Studio であればフォントサイズの変更や、異常系コードの非表示ができることを検証した。ラインハイライトば、eclipse と emacs で実装できることを確認している。

電通大 寺田：プログラムスライシングみたいに、実行可能な形で抜き出せれば、それを使うと正常データだけはちゃんと動くプログラムができるのではないか？

それは面白い。gcc の gcov によるコードカバレッジは、アセンブリ命令のレベルで取れているので、実行された命令と行番号の対応が取れている。そのためプログラムスライスはできると思う。コードカバレッジに中括弧がカウントされないが、これは中括弧が命令に翻訳されないため、カウントが行われていないためだと考えられる。コンパイル時にベーシックブロックに展開されているので、実行された命令とベーシックブロックから中括弧を復元すればプログラムスライスはできると思う。

電通大 岩崎：エラー処理でしか使わないクラスとかあるので、そのような不要なクラスなどを取り除くことはできないか？

ヘッダファイルまで考えが及んでなかった。実際に利用されているクラスや構造体だけ表示する機能は確かに自分もほしい。インテリセンスの候補から削除されると嬉しい。