日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

動的なコードの評価機構を備えた言語に対するテストカバレッジ測定ツール

坂本 一憲　鷲崎 弘宜　深澤 良彰

動的プログラミング言語の普及と共に，メタプログラミングによるコードの動的評価の利用が増加している，しかし，既存のテストカバレッジ測定ツールでは，動的に評価されたコードを測定対象として含むことができない．そのため，Ruby on Rails のようなコードの自動生成と動的評価を利用するようなアプリケーションに対して，ソフトウェアテストの十分性を判断する指標としてテストカバレッジが正常に機能しないことがある．そこで，本論文では動的に評価されたコードを測定対象として含むことができるテストカバレッジの測定手法を提案する．その上で，動的なコードの評価機構を備えたプログラミング言語である Ruby 言語に対応したテストカバレッジ測定ツールの実装方法と構築結果を報告する．

1　はじめに

ソフトウェアテスト（以降，テスト）とは，ソフトウェア開発において混入した欠陥を発見する行為である．社会におけるソフトウェアの重要性が高まり，欠陥のないソフトウェアが強く求められている．そのため，テストは必要不可欠な開発工程となっている．

テストカバレッジ（以降，カバレッジ）とは，テスト対象となるプログラムの全体に対して，プログラムがテストされた割合を示す指標である．カバレッジはテストが十分に実行されたか判断する基準となる [1]．テストカバレッジにはいくつかの種類があり，全てのステートメントにおいて，少なくとも一回以上実行された割合を示すステートメントカバレッジ，全ての条件分岐において，各分岐先が少なくとも一回以上実行された割合を示すデシジョンカバレッジ，全ての条件分岐において，各条件分岐の条件式を構成する全ての論理項が少なくとも一回以上はそれぞれ真と偽の両方の値に評価された割合を示すコンディションカバレッジ，全ての条件分岐において，デシジョンカバレッジとコンディションカバレッジの両方が網羅したと判定された割合を示すコンディション/デシジョンカバレッジなどが存在する．

動的プログラミング言語（以降，動的言語）とは，一般的なプログラミング言語がコンパイル時に決定するような要素を実行時に決定するようなプログラミング言語である．近年では，Ruby，Python，Groovy といった言語の普及と共に，動的言語が脚光を浴びつつある．本論文では特に，動的言語が持つ動的なソースコードの評価機構，多くの場合 `eval` サブルーチンとして提供される機能について論じる．

近年，インターネットの発展と共に，Web アプリケーションの普及が進んでいる．急速に進む Web アプリケーションの需要に対応するために，Web アプリケーションフレームワークの開発が盛んである．フレームワークとは再利用可能なソフトウェアアーキテクチャであり，類似する複数のアプリケーションに対して汎用的な設計を提供する．フレームワークは半完成のアプリケーションであり，開発者がアプリケーション固有のコードを加えることで，アプリケーションを開発できる [2]．例えば，Ruby 言語で開発された `Ruby on Rails` では，Ruby 言語の持つ動的言語の性質を上手く利用して，多くのコードを動的に生成して評価することで，ユーザーが記述すべきコードの量

A Tool for Measureing Test Coverage Metric for Programming Languages with Dynamic Evaluation

Kazunori Sakamoto　Hironori Washizaki　Yoshiaki Fukazawa, 早稲田大学, Waseda University.

を大幅に削減している．

動的言語が持つ `eval` サブルーチンのような動的なコードの評価機構を利用した場合，既存のカバレッジ測定ツール（以降，測定ツール）では動的に評価されたコードを測定対象に含められない．これは，カバレッジの測定対象をプログラム実行前の静的解析のみによって決定していることに起因する．そのため，現状では，動的に評価されたコードに対して，テストが十分であるか判断する指標としてカバレッジが上手く機能していないという問題がある．

本論文では，動的に評価されたコードも測定対象に含められるカバレッジ測定手法を提案する．その上で，実際に Ruby 言語に対応した測定ツールを実装して，ツールのアーキテクチャや実装方法について構築結果を報告する．なお，実装した測定ツールはオープンソースとして公開している．[3]

2 関連研究

Kim らは [5]，静的解析の一種である抽象解釈を利用することで，二段階式の言語，すなわち，動的に評価されたコードの中でさらに動的評価を行わない言語に対して，新しいカバレッジを提案している．これは，本論文が問題とするところの，動的に評価されたコードを測定対象に含むカバレッジである．例えば，Kim らによって提案されたデシジョンカバレッジ $NDC(T)$ は以下のように定義されている．テストケースの集合を T，テストケース t を実行したときに網羅した条件分岐の数を $\#B_t$，ソースコード中に含まれている条件分岐の数を $\#B_c$，動的に評価されたコード中に含まれている条件分岐の数を $\#B_e$ と置くと，

$$NDC(T) = \frac{\sum_{t \subset T} \#B_t}{\#B_c + \#B_e}$$

である．Kim らは，抽象解釈を利用したカバレッジの測定手法について提案しているが，実際に測定するためのアルゴリズムやツールにまでは至らず，今後の展望にて言及するのに留まっている．また，応用可能としながらも，動的評価を二段階までに制限している点，デシジョンカバレッジのみについて言及している点，実際に測定を行うためには不十分な点がある．本論文では，Kim らの研究成果を踏まえて，二段階に制限されないカバレッジ測定手法を提案して，Ruby 言語に対応した四種類のカバレッジを測定可能なツールを実装した．

我々の過去の研究では [4]，複数プログラミング言語対応のカバレッジ測定フレームワーク，題して Open Code Coverage Framework(OCCF) を提案している．OCCF では，抽象構文木もしくは具象構文木を介することで，測定対象のソースコードにカバレッジ測定用の特殊なコード（以降，測定コード）を埋め込み，得られたソースコードをテスト時に実行することで，カバレッジ測定結果の算出に必要な情報を収集して測定を行う．ソースコードに測定用コードを埋め込む方法はそれ以外の方法，例えば，中間言語に埋め込む方法や処理系に測定機能を追加する方法と比べて，実装が容易で開発コストが少ない点で優れていることが分かっている．本論文では，我々の過去の研究成果を踏まえて，ソースコードに測定用コードを埋め込むことで，動的に評価されたコードを測定対象に含んだカバレッジの測定手法を提案する．

3 動的に評価されたコードを測定対象に含まないカバレッジ測定手法

本論文が提案するカバレッジ測定手法の基本的なアイデアは，測定対象の要素が実行される度に，当該要素が実行された旨を記録していき，最終的に測定対象の要素一覧と一回以上実行された要素一覧から，カバレッジ測定結果を算出するというものである．実行された旨を記録するために，測定対象の各要素に対して測定用コードを埋め込む．記録した情報を解析することで，既存の測定ツールと同様にカバレッジを測定できる．

測定対象の要素とは，例えば，ステートメントカバレッジであればステートメント，デシジョンカバレッジであれば条件分岐，コンディションカバレッジであれば論理項である．上述した三種類のカバレッジについて以下でそれぞれ議論する．なお，コンディション/デシジョンカバレッジは，デシジョンカバレッジとコンディションカバレッジの両方を組み合わせれば算出可能であるので割愛する．

3.1 ステートメントカバレッジ

ステートメントカバレッジは各ステートメントが一回以上実行されたか否かで網羅の判定を行う．そのため，各ステートメントに当該ステートメントが実行されたという記録を行う処理を追加すれば良い．そこで，各ステートメントに一意に対応する数値を与えておき，その数値を記録する処理をサブルーチン化する．処理を追加するためにサブルーチン呼び出しを適切に埋め込めばよいが，埋め込み方法は二通りある．一つ目は，各ステートメントの直後に新しいステートメントとしてサブルーチン呼び出しを追加する方法である．二つ目は，各ステートメントをサブルーチンの引数として，覆う形でサブルーチン呼び出しを追加する方法である．前者は，各ステートメントの評価結果の値に関わらず追加できるが，C 言語などにおけるブロックを持たない省略形の if 文の中のステートメントに対して，新たにステートメントを追加する場合，ブロックを補うといった特殊な処理が必要であるという欠点がある．後者は，前者のようにブロックを補う必要はないが，どのようなステートメントの評価結果でも受け取れるようにサブルーチンの引数を設計しなければならないという欠点がある．

```
1 def sample1()
2   p "statement 1"
3   p "statement 2"
4 end
```

図 1 Ruby 言語による測定対象のソースコード

```
1 def sample1()
2   p "statement 1"
3   stmt_cov(1)
4   p "statement 2"
5   stmt_cov(2)
6 end
```

図 2 一つ目の方法で埋め込んだソースコード

例として，Ruby 言語において，測定対象のソースコードを図 4，前者の方法で埋め込んだソースコードを図 5，後者の方法のものを図 6 で示す．なお，`stmt_cov` は各ステートメントに一意に対応する数値を記録するメソッドである．

```
1 def sample1()
2   stmt_cov(1, p "statement 1")
3   stmt_cov(2, p "statement 2")
4 end
```

図 3 二つ目の方法で埋め込んだソースコード

Ruby 言語のステートメントは全て評価結果を返す式である上，動的言語であり，どのような型の値でも引数として受け取れるため，構築した測定ツールでは特殊な処理が不要な後者の方法を採用している．

3.2 デシジョンカバレッジ

デシジョンカバレッジは，各分岐先が少なくとも一回以上実行されたか否かで網羅の判定を行う．そのため，各条件分岐の各分岐先に当該の分岐が行われたという記録を行う処理を追加すれば良い．そこで，ステートメントカバレッジと同様に，各条件分岐に一意に対応する数値を与えておき，その数値を記録する処理をサブルーチン化する．処理を追加するためにサブルーチン呼び出しを適切に埋め込めばよいが，埋め込み方法は二通りある．一つ目は，各分岐先の最初のステートメントにサブルーチン呼び出しを追加する方法である．二つ目は，条件分岐の条件式の値を記録する方法である．前者はステートメントカバレッジにおけるサブルーチン呼び出しを新しいステートメントとして追加する方法と同じ着想である．そのため，同様に省略形 if 文に対してブロックを補うといった特殊な処理が必要となる欠点がある．さらに，else 文のない if 文のように分岐先が省略されている場合，else 節を追加して明示的に分岐先を追加する必要があるという欠点もある．後者は条件式の評価値が必ず真もしくは偽の二値を取ることを利用して，その評価値を記録することで分岐先の記録を行う方法である．この場合，真と偽の両方の値を取れば当該条件分岐において全ての分岐が行われたと見なせる．

```
1 def sample2()
2   if ARGV.count > 1 then
3     p "ARGV.count > 1"
4   end
5 end
```

図 4　Ruby 言語による測定対象のソースコード

```
1 def sample2()
2   if ARGV.count > 1 then
3     branch_cov(1, true)
4     p "ARGV.count > 1"
5   else
6     branch_cov(1, false)
7   end
8 end
```

図 5　一つ目の方法で埋め込んだソースコード

```
1 def sample2()
2   if branch_cov(1, ARGV.count > 1) then
3     p "ARGV.count > 1"
4   end
5 end
```

図 6　二つ目の方法で埋め込んだソースコード

例として，Ruby 言語において，測定対象のソースコードを図**??**，前者の方法で埋め込んだソースコードを図**??**，後者の方法のものを図**??**で示す．なお，branch_cov は各条件文に一意に対応する数値と条件式の評価値を記録するメソッドである．

ステートメントカバレッジと同様に，構築した測定ツールでは特殊な処理が不要な後者の方法を採用している．

3.3　コンディションカバレッジ

デシジョンカバレッジは，各条件分岐の条件式を構成する全ての論理項が少なくとも一回以上はそれぞれ真と偽の両方の値に評価されたか否かで網羅の判定を行う．そのため，各論理項の評価値の記録を行う処理を追加すれば良い．そこで，デシジョンカバレッジと同様に各論理項に一意に対応する数値を与えておき，その数値を記録する処理をサブルーチン化する．処理を追加するためにサブルーチン呼び出しを適切に埋め込めば良く，埋め込み方法は一通りのみである．デシジョンカバレッジの後者の方法と同様に，各論理項の評価値を記録すれば良い．

```
1 def sample3(b1, b2)
2   if b1 && b2 then
3     p "both b1 and b2 are true"
4   end
5 end
```

図 7　Ruby 言語による測定対象のソースコード

```
1 def sample3(b1, b2)
2   if cond_cov(1, b1) && cond_cov(2, b2) then
3     p "both b1 and b2 are true"
4   end
5 end
```

図 8　測定用コードを埋め込んだソースコード

例として，Ruby 言語において，測定対象のソースコードを図 7，測定用コードを埋め込んだソースコードを図 8 で示す．なお，cond_cov は各論理項に一意に対応する数値と論理項の評価値を記録するメソッドである．

4　動的に評価されたソースコードを測定対象として含むカバレッジ測定手法

本節では，前節の手法を改良することで，動的に評価されたコードを測定対象に含む手法を説明する．動的にコードを評価する機構として，ソースコードを文字列で受け取って，それを評価して実行する eval サブルーチンを考える．受け取ったソースコードの中でさらに eval サブルーチンを呼んでいる場合も考慮して，再帰的に呼び出す際の深さに制限を与えないとする．

測定手法のアイデアとして，eval サブルーチンが呼び出されるたびに，動的に評価されたコードに対しても測定用コードを埋め込めば測定対象に含むこと

ができる．前節で説明した測定用コードはテスト時に実行され，測定結果の算出に必要な情報を記録する．同じタイミングで eval サブルーチンに渡されるソースコードに対して測定用コードを埋め込むことができる．すなわち，予め測定用コードの埋め込み処理を測定対象のソースコードに埋め込んでおき，eval サブルーチンが実行される度に，動的に評価されたコードに対しても測定用コードを埋め込む．

そこで，測定用コードを埋め込む処理をサブルーチン化しておき，前節と同様にサブルーチン呼び出しを埋め込めば良い．埋め込み方法は一通りで，各 eval サブルーチンの引数に対してサブルーチン呼び出しを埋め込めばよい．

```
1 def sample4(a)
2   eval(%Q{
3     if #{a} == 0 then
4       eval('p "a is 0"')
5     end
6   })
7 end
```

図 9　Ruby 言語による測定対象のソースコード

```
1 def sample4(a)
2   eval(eval_cov(1, %Q{
3     if #{a} == 0 then
4       eval('p "a is 0"')
5     end
6   }))
7 end
```

図 10　測定用コードを埋め込んだソースコード

例として，Ruby 言語において，測定対象のソースコードを図 7，測定用コードを埋め込んだソースコードを図 10 で示す．なお，eval_cov は文字列で表現された動的に評価されるソースコードに対して，測定用コード（stmt_cov，branch_cov，cond_cov と eval_cov メソッド）を埋め込むサブルーチンである．

図 9 の sample4 メソッドに引数 0 を与えた場合，一つ目の eval を評価すると図 11 のようなソースコードになり，二つ目の eval を評価すると図 12 のようなソースコードになる．

```
1 def sample4(a)
2   if 0 == 0 then
3     eval('p "a is 0"')
4   end
5 end
```

図 11　一つ目の eval を評価した後のソースコード

```
1 def sample4(a)
2   if 0 == 0 then
3     p "a is 0"
4   end
5 end
```

図 12　二つ目の eval を評価した後のソースコード

同様に，図 10 の sample4 メソッドに引数 0 を与えた場合，一つ目の eval を評価すると図 13 のようなソースコードになり，二つ目の eval を評価すると図 14 のようなソースコードになる．

```
1 def sample4(a)
2   stmt_cov(2, if branch_cov(4, 0 == 0) then
3     stmt_cov(3, eval(eval_cov(5, 'p "a is 0"')))
4   end)
5 end
```

図 13　一つ目の eval を評価した後の測定用コードが埋め込まれたソースコード

```
1 def sample4(a)
2   stmt_cov(2, if branch_cov(4, 0 == 0) then
3     stmt_cov(3, stmt_cov(6, p "a is 0")))
4   end)
5 end
```

図 14　二つ目の eval を評価した後の測定用コードが埋め込まれたソースコード

Ruby 言語には動的にコードを評価する機構として，eval，insntace_eval，class_eval メソッドの三種類が存在する．そこで，構築した測定ツールで

は，この三種類のメソッドに対して eval_cov メソッドを埋め込む．

5 Ruby 言語に対応したカバレッジ測定ツール

我々は第 3 節と第 4 節で述べた手法を用いて，Ruby 言語に対応した測定ツールの構築を行った．測定ツールは，ソースコードに測定用コードを埋め込み，そのソースコードをテスト時に実行することで，カバレッジに必要な情報を記録する．また，動的にコードを評価する三種類のメソッドに対しても測定用コードを埋め込むことで，動的に評価されるコードに対しても実行時に測定用コードを埋め込む．そのため，動的に評価されたコードも含めたカバレッジを測定できる．

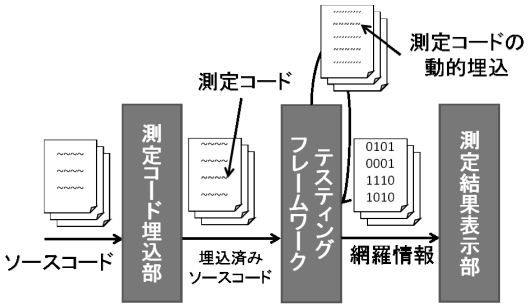

図 15　測定ツールのアーキテクチャ

測定ツールのアーキテクチャを図 15 で示す．測定ツールは測定コード埋込部，テスティングフレームワーク，測定結果表示部の三機能部によって構成されている．

測定ツールによるカバレッジの測定手順は以下の通りである．

1. **カバレッジ測定コードの埋め込み**
   カバレッジを測定するための特殊なコードをソースコードに埋め込む．
2. **テストの実施**
   測定用コードが埋め込まれたソースコードに対してテストを実施することで，測定に必要な情報を記録する．また，動的に評価されたコードに対しても逐次測定用コードを追加していく．
3. **測定結果の表示**
   記録した情報を解析してカバレッジの測定結果を表示する．

以下で，三機能部についてそれぞれ詳細を述べる．

5.1 測定コード埋込部

測定コード埋込部では，四種類のカバレッジを測定するために，stmt_cov，branch_cov，cond_cov メソッドをソースコードに埋め込む．また，動的に評価されるコードも測定対象に含むために，eval_cov メソッドを eval，insntace_eval，class_eval メソッドの引数に埋め込む．こうした操作は Ripper2Ruby [6] を利用して抽象構文木を介して行う．Ripper2Ruby は Ruby1.9 以降に対応したライブラリであり，Ruby のソースコードと抽象構文木の相互変換を実現する．また，抽象構文木に対するノードの探索，追加，削除，置換といった操作を提供しており，測定コード埋込部はこのライブラリを利用することで実装している．

5.2 テスティングフレームワーク

テスティングフレームワークは測定ツールが持つ機能ではなく，Java 言語であれば JUnit，Ruby 言語であれば標準ライブラリの Test::Unit や RSpect などの外部のアプリケーションが該当する．測定用コードを埋め込んだソースコードをテスティングフレームワークで実行することで，自動的にカバレッジの測定に必要な情報が記録され，また，動的に生成されたコードに対して実行時に測定用コードが埋め込まれる．

5.3 測定結果表示部

測定結果表示部は測定用コードが記録した情報を利用してカバレッジの測定結果を表示する．測定用コードは各測定要素に一意に対応する数値をファイルに記録しているが，プロセス間通信や TCP/IP を利用するなど，様々な記録方法が考えられる．測定結果表示部では測定結果をコンソール画面に出力しているが，GUI を利用した表示や XML ファイルへの出力など，機能を拡張することが可能である．

6 まとめと今後の展望

ソースコードに測定用コードを埋め込むことでカバレッジを測定できることを説明し，さらに，同様の手法で動的に評価されるコードを測定対象に含むことができることを説明した．その上で，Ruby 言語に対応した測定ツールの実装を示した．今後はパフォーマンスの最適化やユーザビリティの向上を行い，`Ruby on Rails` といった大規模なソフトウェアに対する適用実験を行う予定である．

謝辞 本研究の一部は早稲田大学グローバル COE プログラムによった．

参 考 文 献

[1] Lee Copeland: A Practitioner's Guide to Software Test Design, Artech House, 2003.

[2] Mohamed Fayad and Douglas C. Schmidt: Object-Oriented Application Frameworks, the Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[3] Kazunori Sakamoto, xrcov, http://github.com/KAZUu/xrcov.

[4] Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa：Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages, 2010 10th International Conference on Quality Software(QSIC), 2010.

[5] Taeksu Kim, Chunwoo Lee, Kiljoo Lee, Soohyun Baik, Chisu Wu and Kwangkeun Yi: Test Coverage Metric for Two-staged Language with Abstract Interpretation, 2009 16th Asia-Pacific Software Engineering Conference(APSEC), 2009, pp. 301–308.

[6] Sven Fuchs and Kristian Mandrup: Ripper2Ruby, http://github.com/kristianmandrup/ripper2ruby.