RB-006

柔軟かつ複数プログラミング言語対応のテストカバレッジ測定フレームワーク
A Flexible Test-Coverage-Measurement Framework Supporting Multiple Programming Languages

坂本 一憲 * 鷲崎 弘宜 * 深澤 良彰 *
Kazunori Sakamoto Hironori Washizaki Yoshiaki Fukazawa

1. はじめに

テストカバレッジ（コードカバレッジ；テスト網羅率）とは，ソフトウェアテストを行う際に用いられる指標である．テスト可能なプログラムの全体に対して，プログラムのソースコードがテストされた割合を意味する．テストカバレッジは，ソフトウェアテストが十分に実行されたかどうかを判断する基準となる．

テストカバレッジにはいくつかのレベルが存在する．例えば，すべ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ステートメントの割合を示すステートメントカバレッジ (C0)，すべ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条件分岐の割合を示すデシジョンカバレッジ (C1)，すべての条件分岐の条件式を構成する論理項において，すべての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論理項の割合を示すコンディションカバレッジ (C2) などである．テストカバレッジレベルは，ソフトウェアテストの目的に応じて適切なレベルが選択される [1].

テストカバレッジ測定ツール（以降，測定ツール）の普及が進み，C 言語や Java 言語といった主要なプログラミング言語（以降，言語）に対して，実用的な有償，無償ツールが提供されている．一方，COBOL 言語といったレガシーな言語や，Lua 言語といったマイナーな言語において，測定ツールが存在していない，もしくは，高価な製品のみが提供されている．その上，パラダイムの変化と言語の多様化が進んでおり，新しい言語への対応や，言語の仕様変更への対応が必要とされている．

近年では，複数の言語を用いたソフトウェア開発の需要が高まっている．クライアントサーバモデルに基づいて設計されたソフトウェアの場合，クライアントアプリケーションとサーバアプリケーションが別々に開発される．このようなアプリケーションは，それぞれ異なる言語で開発されるケースがある．そういったソフトウェアをテストする場合，モジュール単体をテスト対象とする単体テストでは問題がないが，モジュールの組み合わせをテスト対象とする結合テストでは，複数の言語を統一的に扱える測定ツールが必要となる．

テストカバレッジは，特定の言語に依存しない概念である．ステートメントや条件分岐といったほぼすべての言語に共通する概念を用いるため，多くの言語のソフトウェアテストにおいて利用できる．しかし，多くの測定ツールは，各言語に特化した設計になっている．そのため，各言語に対してそれぞれ測定ツールが開発されており，そういった測定ツールの新規開発，保守のコストは莫大なものとなっている．また，上述したような複数の言語で開発されたソフトウェアを測定するためには，複数の測定ツールを組み合わせる必要がある．しかし，複数の測定ツールを組み合わせると，対応するテストカバレッジレベルの差異や，実装の差異による測定基準の不一致のため，テストカバレッジの有効性が失われる場合がある．

そこで我々は，複数言語対応のテストカバレッジ測定フレームワーク（以降，提案 FW）を提案する．提案 FW では言語のシンタックスに焦点を当てることで，言語間の共通要素の抽出と相違点の吸収を行い，特定の言語に特化した測定ツールの開発コストを低減する．さらに，新規言語への対応や機能拡張の支援を行い，複数の言語における統一的で柔軟なテストカバレッジを実現する．

なお，研究成果である提案 FW はオープンソースとして公開しており，入手して利用できる [2].

2. 既存の測定ツールにおける問題点

既存の測定ツールにおける問題点を以下で述べる．

2.1 新規開発コストの高さ

多くの測定ツールは，レガシーな言語やマイナーな言語に対応していない，もしくは，対応するツールは非常に高価である．例えば，我々が調査した限りでは，COBOL 言語に対応する無償の測定ツールが存在しなかった．さらに，言語の多様化も進んでおり，測定ツールの新規開発に対する需要が高まっている．

測定ツールは，ソースコードを解析して，ソフトウェアテストを行う際にカバレッジの情報を収集して，測定結果を表示する．測定ツールは，ソースコードから構文を解釈する構文解析器，ステートメントや条件分岐といった構文のセマンティクスを解釈する意味解析器，カバレッジの測定機能と測定結果の表示機能から構成される．これらの機能は一般的に実装が難しく，開発に莫大なコストが必要とされる．特に，測定結果の表示機能以外は，特定の言語への依存性が強く，再利用による新規開発コストの低減が困難である．

* 早稲田大学 大学院 基幹理工学研究科

2.2 保守コストの高さ

言語のパラダイム変化や機能拡張のため，言語仕様が変化することがある．そして，言語仕様の変化に伴って，構文の種類が増減したり構文のセマンティクスが変更される．例えば，Java 言語が 1.4 から 5.0 にバージョンアップした際，拡張 for 文や総称型などの新しい構文と概念が言語仕様に追加された [3]．

測定ツールが言語仕様の変化に対応する場合，新しい構文に対応するため，構文解析器と意味解析器の修正が必要であり，場合によっては，カバレッジの測定機能も修正する必要がある．そのため，測定ツールの保守は必要であり，そのコストは莫大なものとなる．

2.3 統一性を欠いた測定

Web アプリケーションの普及などに伴って，複数の言語を用いたソフトウェア開発が広まっている．例えば，クライアントサーバモデルに基づいてソフトウェアを設計した場合，クライアントアプリケーションを Python 言語で開発して，サーバアプリケーションを Java 言語で開発するケースが考えられる．

複数の言語で開発されたソフトウェアの場合，結合テストにおけるカバレッジ測定が困難である．用いた言語すべてに対応したツールがなければ，複数のツールを組み合わせる必要がある．しかし，複数のツールでは測定に手間がかかる上，対応するテストカバレッジレベルの差異や測定基準の差異の影響を受ける．そのため，ソフトウェアテストに対する指標として有効性が失われる場合がある．

我々が調査した無償の測定ツールの中では，GNU Compiler Collection のサブセットとして提供される gcov [4] 以外に，複数の言語に対応したツールは存在しない．

2.4 柔軟性を欠いた測定

テストカバレッジはソフトウェアテストが十分であるか評価する有効な指標である．テストカバレッジが 100% を満たすことで，ソフトウェアテストが十分であることが明確に判断できる．しかし，開発者が必要であると判断した部分のみテストできれば，たとえテストカバレッジが 100% 未満であってもソフトウェアテストは十分である．このように，テストカバレッジは割合であり定量的な結果を示す．そのため，100% 未満の測定結果からは，かえって定性的な判断が困難であり，必要なソフトウェアテストを完遂できたかどうか評価することが難しい．

また，プログラム内のすべての要素を測定対象とすることは，効率性の面からも問題が指摘されており，測定範囲を限定する必要性がある．例えば，坂田ら [5] は，コンポーネントに対するカバレッジの測定において，利用する機能のみを測定対象とする手法を提案している．

我々が調査した測定ツールや研究には，カバレッジ測定範囲や測定対象要素を柔軟に指定できるものは存在しない．

2.5 不完全な測定の可能性

テストカバレッジは，ソフトウェアテストを実施した際に得られた情報に基づいて測定される．しかし，実行可能なバイナリファイルに対して測定する場合，ソースコードと実行可能なバイナリファイルのセマンティクスの違いから，ソースコード上に存在する要素であっても，測定されない場合がある．

例えば，Java 言語の実行可能なバイナリファイルであるクラスファイルでは，ソースコード上のどこからも呼び出されない private メソッドや，常に条件式が偽となる if 文内のデッドコードは削除されることが多い．そのため，クラスファイルに対して測定すると，デッドコードが測定対象に含まれない．

Java 言語用の測定ツールである Cobertura[6] では，この手法を用いるため，デッドコードに対する測定ができない．

3. カバレッジ測定フレームワークの提案

我々は，複数言語対応のカバレッジ測定フレームワークを提案し，上述の問題を解決，および緩和する．

フレームワーク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 [7]．

3.1 全体の仕組み

提案 FW の全体像と処理の流れを図 1 で示す．図 1 のように，提案 FW は，コード埋め込み，コード実行，カバレッジ表示の三つのサブシステムから構成される．さらに，コード埋め込みサブシステムは，AST（Abstract Syntax Tree；抽象構文木）生成部，AST 整形部，AST 操作部，コード生成部の四つの機能要素から構成される．

カバレッジ測定の流れを以下に示す．

1. ソースコードから AST の生成
2. AST を介した測定用コードの埋め込み
3. AST からソースコードの生成
4. 生成されたソースコードの実行と情報収集
5. テストカバレッジの測定結果の表示

提案 FW では測定対象のソースコードに測定用コードを埋め込み，そのプログラムを実行することでテストカバレッジを測定する．測定用コードを埋め込む際に，ソースコード中における測定対象要素の位置情報など，測定結果として表示する際に必要となる情報も収集する．コンパイラの最適化機能がデッドコードなどを削除する前に測定用コードを埋め込むため，ソースコード上に存在する全ての要素を完全に測定することができる．

提案 FW の有用性を図 2 で示す．提案 FW は，オブジェ

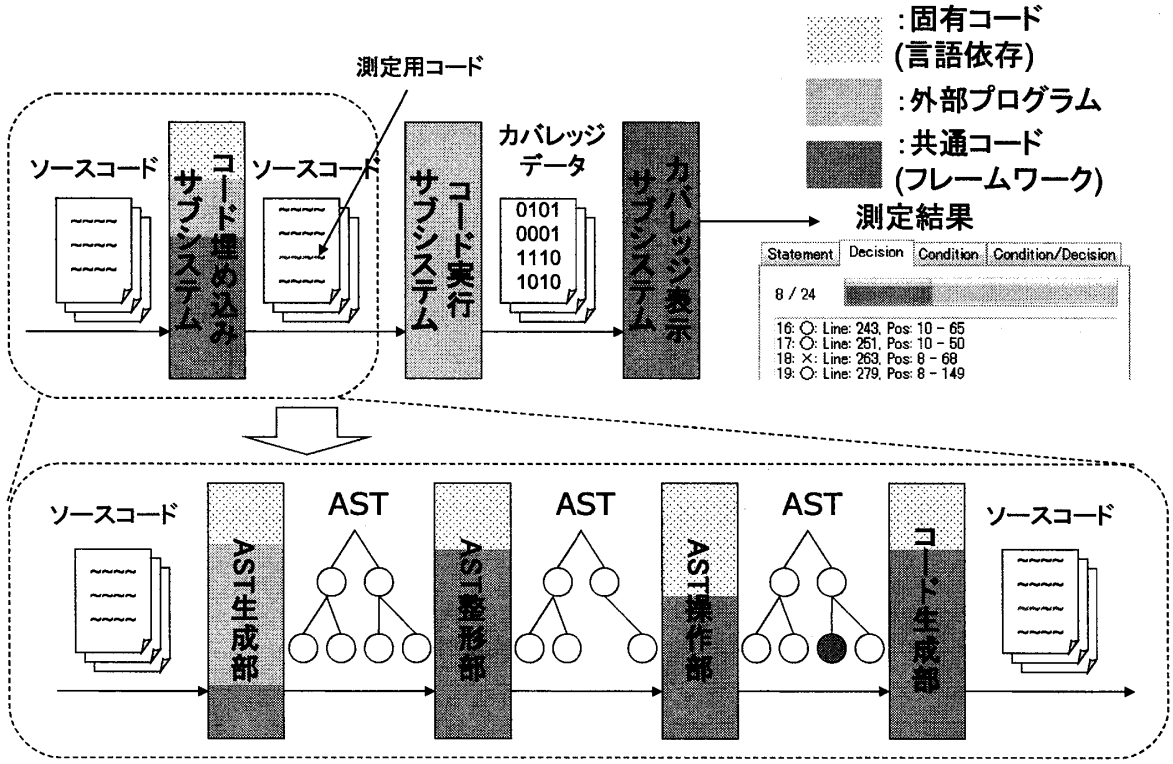

図 1　提案フレームワークの概観

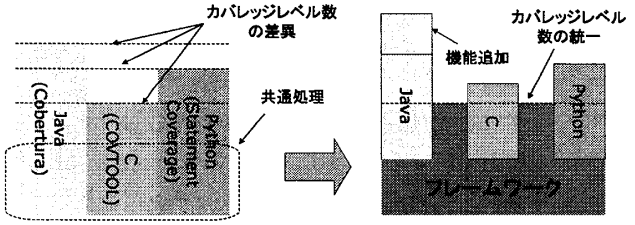

図 2　提案フレームワークの有用性

クト指向プログラミングの仕組みを用いて，オブジェクト指向フレームワークとして設計されている．そのため，以上に述べたような機能に対して，言語非依存処理に該当する共通コードの提供と，言語依存処理に該当する固有コードの実装を簡易化するための構造を提供できる．また，ソースコードの編集以外にもプラグインアーキテクチャやスクリプト言語に対応しており，固有コードの追加による機能拡張を容易にする．これにより，柔軟な測定範囲の変更や，特殊なカバレッジの測定を支援する．さらに，カバレッジレベル実装に関するスケルトンコードを提供することで，カバレッジレベルの種類数の統一化を促すガイドラインを与える．現時点では，ステートメントカバレッジ，デシジョンカバレッジ，コンディションカバレッジ，デシジョン／コンディションカバレッジの四種類のカバレッジレベルへの対応を支援する．なお，パスカバレッジのようなソースコードのセマンティクスの解析が必要なカバレッジ以外であれば，簡単な固有コードの追加のみで対応することが可能である．

測定用コードを埋め込む操作は AST 上で行う．AST 上で埋め込むことで，各言語における埋め込み処理の相違点を減らし，共通コードとして抽出しやすくする．さらに，各言語のシンタックスに焦点を当てているため，言語仕様の変化が起こった際の影響を最小限に抑える．これにより，新規開発や保守のコストを低減する．ただし，測定用コードを埋め込むことで測定するという仕組み上，手続き型言語のみを対象とする．

測定用コードはカバレッジ情報の収集処理以外は行わない．提案 FW は，収集処理を行うという点を除いて，測定対象のソースコードのセマンティクスが変わらないように埋め込む．そのため，ソフトウェアテストにおける振る舞いは変化しない．

以下で，リスト 1 に埋め込み前，リスト 2 に埋め込み後のソースコードを示し，測定用コードの埋め込み例を示す．

リスト 1　埋め込み前

```
1  int main() {
2    int a = 0;
3    printf("test");
4    if (a == 0) {
5      puts("a == 0");
6    }
7    else {
8      puts("a != 0");
9    }
10 }
```

リスト 2　埋め込み後

```
1  int main() {
2    int a = 0;
3    statement_coverage(0);
4    printf("test");
5    if (decision_coverage(0, a == 0)) {
6      statement_coverage(1);
7      puts("a == 0");
8    }
9    else {
10     statement_coverage(2);
11     puts("a != 0");
12   }
13 }
```

例中の statement_coverage サブルーチンは，ステートメントカバレッジの測定に，decision_coverage サブルーチンは，デシジョンカバレッジやコンディションカバレッジの測定に利用する．decision_coverage サブルーチンは値渡しで条件式，もしくは論理項の評価値を受け取って，そのまま戻り値として返す．したがって，埋め込み前と後のコードをどちらも同じソフトウェアとしてテストできる．

4. カバレッジ測定フレームワークの実装

提案 FW は .NET Framework 3.5 SP1[8] で実装した．提案 FW では固有コードを追加するために以下の三つの方法に対応している．

- フレームワークのソースコードの直接編集
- アセンブリファイルの作成と配置
- スクリプトファイルの作成と配置

提案 FW では .NET Framework 3.5 SP1 とそれ以前のバージョンで動作するアセンブリファイルと，Dynamic Language Runtime[9]（以降，DLR）が対応するスクリプト言語で記述したスクリプトファイルを読み込んで実行することが出来る．これは，Managed Extensibility Framework[10]（以降，MEF）を用いており，提案 FW が提供するインタフェースを実装するクラスに対して，MEF が提供する属性を付加することで，そのクラスのインスタンス生成を自動化できる．これにより，プラグインアーキテクチャを構築して，アセンブリファイルやスクリプトファイルを自動的に読み込む．なお，DLR も MEF も .NET Framework 上で動作するライブラリである．

本節では実際に提案 FW を用いて，Java 言語，C 言語，Python 言語に対して，ステートメントカバレッジ，デシジョンカバレッジ，コンディションカバレッジ，デシジョン／コンディションカバレッジの四種類のレベルのテストカバレッジを測定するソフトウェアを実装した例を用いて，提案 FW の詳細について述べる．

4.1 コード埋め込みサブシステム

コード埋め込みサブシステムは，以下で述べるような四つの機能要素によって構成される．

- AST 生成部

AST 生成部は，ソースコードを入力して XML フォーマットの AST を出力する．実装例では，Java 言語では SableCC[11]，C 言語では ANTLR[12]，Python 言語では標準ライブラリの AST 生成機能を用いて，外部プログラムとして実装する．AST 生成部はこれらの外部プログラムを呼び出すことで実装される．そのため，外部プログラムの呼び出し処理を共通コードとして Template Method パターン [13] を用いて設計しており，容易に再利用できる形で提供する．Template Method パターンとは，抽象クラス内で自身の抽象メソッドを呼び出す処理を定義して，処理の枠組みを決めておき，その抽象クラスを継承した子クラスで抽象メソッドを実装することで，具体的な処理内容を決定するというデザインパターンである．

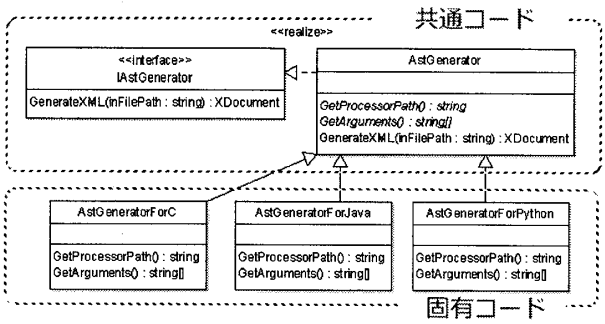

図 3　AST 生成部のクラス図

AST 生成部に関連するクラスとそれらの関係を図 3 の UML[14] クラス図で示す．図 3 の AstGenerator 抽象クラスが，Template Method パターンにより設計されたクラスである．これを利用した例として，Java 言語向けの AST 生成部の固有コードをリスト 3 で示す．リスト 3 の 15 行目の外部プログラム名と 19 行目の引数の指定のように，外部プログラムを呼び出すパラメータの記述のみで，簡単に実装できる．

リスト 3　AstGeneratorForJava.cs

```
1  using System.ComponentModel.Composition;
2
3  namespace CoverageFramework.AstGenerator.Java
4  {
5    [Export(typeof(IAstGenerator))]
6    public class AstGeneratorForJava : AstGenerator
7    {
8      private static readonly string[]
9        _arguments = new[]
10     {
11       "-jar",  "../Java/Java.jar",
12     };
13     protected override string FileName
14     {
15       get { return "java"; }
16     }
17     protected override string[] Arguments
18     {
19       get { return _arguments; }
20     }
21   }
22 }
```

- AST 整形部
  AST 整形部は，AST を AST 操作部で操作しやすいように変形する．
  実装例では入力された AST の不要なノードの削除や新たなノードの追加を行う．具体的には，子要素として非終端記号を一つだけ持つ非終端記号を削除する処理，一行で記述可能な if 文の省略形などを完全な形に変形する処理である．前者は共通コードとして完全に提供されるが，後者は if 文の省略形の特定と変形方法を固有コードとして実装する必要がある．なお，そのコードは AST 操作部を用いて容易に実装できる．
- AST 操作部
  AST 操作部は，様々な機能の集合体である．具体的には，AST が木構造であることを利用して，部分木を列挙する機能，部分木を生成する機能，AST 中の部分木を入力として新たな部分木に置換する機能がある．
  部分木を列挙する機能は，測定用コードの埋め込み位置の特定に利用する．例えば，Python 言語でコンディションカバレッジの測定を実現するために，条件式を構成する論理演算子に注目して分解したすべての項に対して，測定用コードを埋め込む．そのためには，条件式に対応する部分木の中から，各論理項に対応する部分木を列挙する処理が必要である．この処理は言語非依存であるため，Template Method パターンによって設計した共通処理の実装を提供する．

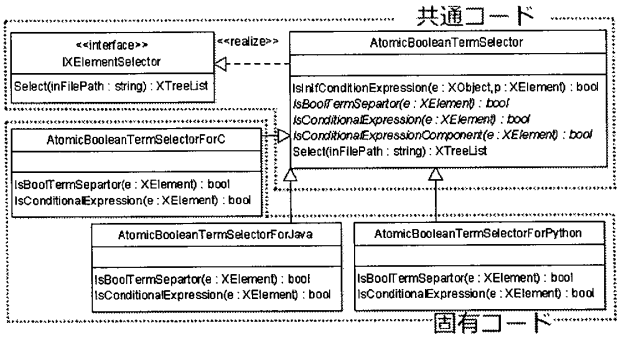

図 4　AST 生成部のクラス図

この処理に関係するクラス図を図 4 で示す．図 4 の `AtomicBooleanTermSelector` 抽象クラスが，Template Method パターンにより設計されたクラスである．これを利用した例として，Python 言語向けの論理項を列挙する固有コードをリスト 4 で示す．リスト 4 の 27 行目のノードが論理演算子かどうかを判定するメソッド，33 行目の条件式かどうかを判定するメソッド，38 行目の論理項かどうかを判定するメソッドのように，与えられたノードが測定対象であるかどうかを判定する処理を記述するだけで，簡単に実装できる．

リスト 4　AtomicBooleanTermSelectorForPython.cs

```
1  using System.Linq;
2  using System.Xml.Linq;
3  using System.ComponentModel.Composition;
4
5  namespace CoverageFramework.Element.Selector.Python
6  {
7    [Export(typeof(IXElementSelector))]
8    public class AtomicBooleanTermSelectorForPython
9      : AtomicBooleanTermSelector
10   {
11     private static readonly string[]
12       _condComponentNames  = new[]
13     {
14       "or_test",  "and_test",
15     };
16     private static readonly string[]
17       _condNames = new[]
18     {
19       "or_test",  "and_test",
20     };
21     private static readonly string[]
22       _condOpValues = new[]
23     {
24       "or",  "and",
25     };
26
27     protected override bool
28       IsBoolTermSepartor(XElement e)
29     {
30       return !e.HasElements &&
31         _condOpValues.Contains(e.Value);
32     }
33     protected override bool
34       IsConditionalExpression(XElement e)
35     {
36       return _condNames.Contains(e.Name.LocalName);
37     }
38     protected override bool
39       IsConditionalExpressionComponent(XElement e)
40     {
41       return _condComponentNames
42         .Contains(e.Name.LocalName);
43     }
44   }
45 }
```

その他にも，複数の列挙結果を一つの列挙結果にまとめる `XElementSelectorUnion` クラスや，列挙結果の中からさらに列挙を行う `XElementSelectorPipe` クラスなど，様々な処理を提供する．これらは，Command パターン [13] によって設計されており，`IXElementSelector` インタフェースを実装したクラスのインスタンスを組み合わせて，新しい処理を行う機能を実現できる．なお，Command パターンとは，処理要求とそれに伴うパラメータをカプセル化してインスタンスで表現するデザインパターンである．特に，Command にあたるインスタンスを組み合わせて作ったインスタンスを Macro Command と呼ぶ．

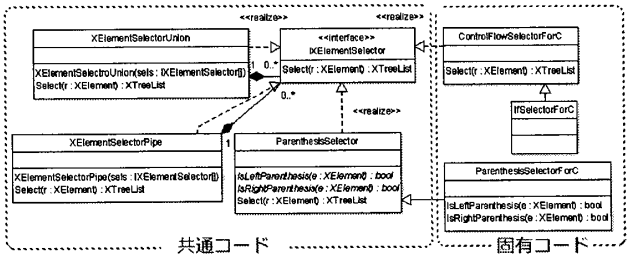

図 5　部分木の列挙に関係するクラス図

部分木の列挙に関係するクラス図を図 5 で示す．図 5 の XElementSelectorPipe クラスが Command パターンによって設計されたクラスである．このクラスは，Macro Command に相当しており，複数の Command インスタンスを組み合わせて新たに一つの処理を表現する．これを利用した例として，条件文に対応する部分木の列挙処理を表す IfSelectorForC クラスのインスタンスと，括弧で括られたコードに対応する部分木の列挙処理を表す ParenthesisSelectorForC クラスのインスタンスを組み合わせて，C 言語向けの条件式に対応する部分木の列挙処理を表すコード断片をリスト 5 で示す．

リスト 5　XElementSelectorPipe の使用例

```
1  var ifSelector = new XElementSelectorPipe(
2    new IfSelectorForC(),
3    new ParenthesisSelectorForC());
```

リスト 5 では，二種類のクラスのインスタンスを与えることで，条件式を列挙するインスタンスを作成している．このように，処理の粒度を小さくして，プログラムの部品化と再利用を促進する．また，このようにして部分木の特定の処理を記述することで，柔軟な測定を実現する．

なお，部分木を生成する機能は，測定コードに対応する部分木の生成に利用される．この機能は固有コードとして実装しなければならない．一方，部分木を置換する機能は，測定コードに対応する部分木を AST に埋め込むために利用される．この機能は共通コードとして完全に提供される．

- コード生成部

コード生成部は，AST から各言語のソースコードを出力する．AST の設計次第ではあるが，各終端ノードがソースコード内のすべてのトークンを属性として記憶している場合，トークンをそのまま出力するだけで良く，各言語間の処理内容にほとんど差異はない．そのため，提案 FW では Template Method パターンによって設計された共通処理の実装を提供する．

コード生成部に関係するクラス図を図 6 で示す．図 6 の SourceCodeGenerator 抽象クラスが，Template Method パターンにより設計されたクラスである．これを利用した例として，Python 言語向けのコード生成部の固有コードをリスト 6 で示す．Python 言語向けの AST の終端ノードでは改行とインデントを属性として記憶していないため，それらを表す終端ノードに対して，別途改行とインデントの出力が必要となる．リスト 6 の 13 から 27 行目までのように，終端ノードに対する例外的な処理を記述するだけで，簡単に実装できる．

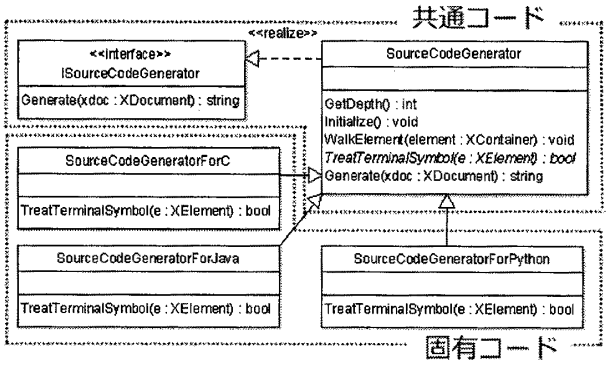

図 6　コード生成に関係するクラス図

リスト 6　SourceCodeGeneratorForPython.cs

```
1  using System.Xml.Linq;
2  using System.ComponentModel.Composition;
3
4  namespace CoverageFramework.CodeGenerator.Python
5  {
6    [Export(typeof(ISourceCodeGenerator))]
7    public class SourceCodeGeneratorForPython
8      : SourceCodeGenerator
9    {
10     protected override bool
11       TreatTerminalSymbol(XElement element)
12     {
13       switch (element.Name.LocalName)
14       {
15       case "NEWLINE":
16         WriteLine();
17         break;
18       case "INDENT":
19         Depth++;
20         break;
21       case "DEDENT":
22         Depth--;
23         break;
24       default:
25         return false;
26       }
27       return true;
28     }
29   }
30 }
```

4.2　コード実行サブシステム

コード実行サブシステムは，測定用コードを埋め込んだプログラムを実行する．提案 FW は，コードを実行する処理系に関する制約がないため，任意の処理系で実行できる．

測定用コードを埋め込んだプログラムを実行することで，得られた情報をカバレッジ表示サブシステムに送信する．送信の仕組みとして，ソケットによる TCP/IP 通信と，ファイル出力の二種類を提供する．

4.3　カバレッジ表示サブシステム

カバレッジ表示サブシステムでは，測定用コードを埋め込む際に得られた情報と，コード実行サブシステムから受信した情報を解析して，測定結果を表示する．

測定用コードを埋め込む際に得られた情報は，位置情報とタグを含む．位置情報は，ソースコードのファイルシステム上のパスと，測定対象要素のソースコードにおける位置を示すために行番号と列番号を持つ．タグは，階層構造を

表す文字列であり，集計したテストカバレッジの結果を表示する際に，フィルタリングを可能にする．例えば，タグで名前空間やクラス階層を表現することで，特定の名前空間やクラス階層のみの結果に限定できる．

測定結果は，位置情報を利用して，測定対象要素がソフトウェアテストの際に実行されたかどうかと，ソースコードにおいてどこに位置するかを示す．

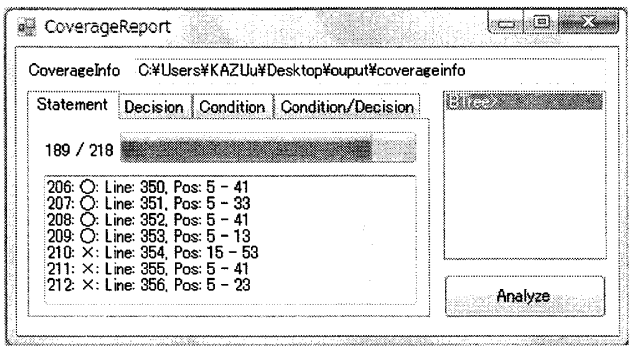

図 7　カバレッジ表示サブシステム

カバレッジ表示サブシステムの表示画面を図 7 で示す．上部のプログレスバーがカバレッジの割合を示しており，中央部のテキストでは，各測定対象要素について，ソフトウェアテストの際に実行されたかどうかの○×とともに，ソースコードにおける位置を行と列で表す．

5.　評価

代表的な測定ツールと比較して，提案 FW の評価を行う．Java 言語では Cobertura, Python 言語では Statement coverage for Python[15] を比較対象とする．

5.1　新規開発コストの低減

測定ツールにおける測定用コードを埋め込む処理に関する部分の LOC (Lines Of Code; コード行数) とカバレッジレベルの種類数を比較して，新規開発コストを評価する．

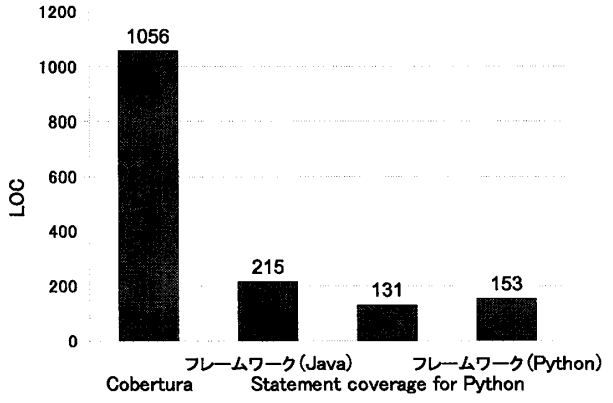

図 8　測定コードの埋め込み処理に関する部分のコード行数

Java 言語に対応する測定ツールと Python 言語に対応する測定ツールにおける，測定用コードを埋め込む処理に関するコードの LOC の比較結果を図 8 で示す．図 8 のように，Java 言語に対応する既存ツールの Cobertura では 1056 行で，提案 FW を用いた実装例の Java 言語特有の埋め込みに関する部分では 215 行である．なお，Cobertura では実行可能なバイナリ形式のクラスファイルに対してコードを埋め込むため，BCEL[16] というライブラリを利用している．しかし，実装例では自作した簡単なヘルパメソッドや標準ライブラリを利用しているが，それ以外のライブラリを一切使用していない．一方，Python 言語に対応する既存ツールの Statement coverage for Python では 131 行で，提案 FW を用いた実装例の Python 言語特有の埋め込みに関する部分では 153 行である．Statement coverage for Python では Python の標準ライブラリを利用している．なお，提案 FW の測定用コードを埋め込む処理に関する共通コードのコード行数は 654 行である．

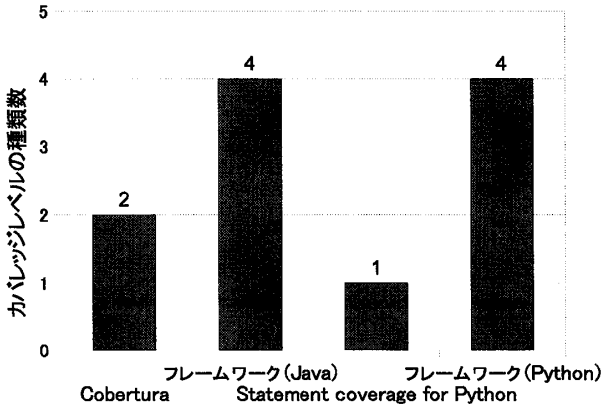

図 9　対応カバレッジレベル数

Java 言語に対応する測定ツールと Python 言語に対応する測定ツールにおける，対応するカバレッジレベルの種類数の比較結果を図 9 で示す．図 9 のように，Cobertura はステートメントカバレッジとデシジョンカバレッジの二種類で，Statement coverage for Python はステートメントカバレッジの一種類にのみ対応している．一方，提案 FW を用いた実装例では，ステートメントカバレッジ，デシジョンカバレッジ，コンディションカバレッジ，デシジョン／コンディションカバレッジの四種類に対応している．したがって，既存ツールより多機能であるのにも関わらず，少ないコード行数で実装が可能となっている．

提案 FW では図 1 で示されるコード埋め込みサブシステムの一部を実装するだけで，新しい言語に対応した測定ツールを開発できる．一方，提案 FW を用いずに新たに開発する場合は，全てのサブシステムに該当するものを開発する必要がある．さらに，コード埋め込みサブシステムの部分のみを比較しても，上述の比較結果のように，複数言

語への対応を前提とした場合，提案 FW を用いた方が実装に必要なコード行数の総量が少なくなる．

以上から，新規開発コストの問題の緩和に成功した．

5.2 保守コストの低減

言語仕様の変更に伴って必要となる修正について議論することで，保守コストを評価する．

言語仕様の変更に伴う修正コストについて評価を行う．Cobertura では BCEL というライブラリを用いてクラスファイルの操作を行っているが，クラスファイルの仕様が変更された場合，BCEL がその変更に対応するまで待つか，もしくは独自にライブラリを修正しなければならない．一方，提案 FW では言語のシンタックスに焦点を当てているため，測定対象要素の特定と測定コードの埋め込みが可能でさえあれば正常に動作する．そのため，図 1 で示される AST 操作部の一部を修正するだけで済む，もしくは，修正が不要な場合もある．

Java 言語が 1.4 から 5.0 にバージョンアップした場合を例に取ると，提案 FW では，AST 操作部に拡張 for 文に対応する AST 上の部分木の列挙処理を追加するだけで良い．一方，Cobertura では，拡張 for 文以外にもジェネリクスなど全ての言語仕様の変化に BCEL が対応した上で，拡張 for 文に対する埋め込み処理を追加する必要がある．したがって，提案 FW は修正する範囲を低減して，コストを削減できる．ただし，構文解析器は提案 FW でも Cobertura でも，コンパイラコンパイラを用いて実装しているため，保守コストに差異は無い．なお，コンパイラコンパイラとは構文解析器のソースコードを自動生成するためのソフトウェアである．また，カバレッジ結果の集計や表示に関しては，どちらも修正が不要である．

以上から，測定ツール全体における保守コストの問題の緩和に成功した．

5.3 統一的な測定

複数の言語で開発されたソフトウェアにおけるカバレッジ測定について議論することで，統一的な測定を評価する．

Java 言語で実装したサーバアプリケーションと，Python 言語で実装したクライアントアプリケーションの両方を稼働して結合テストを行う場合，サーバアプリケーションのカバレッジ測定に Cobertura を用いて，クライアントアプリケーションのカバレッジ測定に Statement coverage for Python を用いるとする．しかし，Cobertura ではステートメントカバレッジとデシジョンカバレッジを測定できるが，Statement coverage for Python ではステートメントカバレッジしか測定できない．Python 言語において，デシジョンカバレッジを測定できる無償ツールは，我々が探した範囲内では存在しなかった．この場合，それぞれのプログラムに対する異なったレベルのテストカバレッジ測定か，もしくは，同じレベルであるステートメントカバレッジの測定かを選ばざるを得ない．そのため，テストの指標として不十分なテストカバレッジしか得られない可能性がある．

一方，提案 FW を用いた場合，それぞれのプログラムに対して，測定基準の差異がない同じレベルのテストカバレッジを測定できる．そのため，テストの指標として有効なテストカバレッジが得られる．

以上から，複数の言語を用いたソフトウェアのカバレッジ測定における問題の解決に成功した．

5.4 柔軟な測定

カバレッジ測定範囲のカスタマイズについて議論することで，柔軟な測定を評価する．

Cobertura も Statement coverage for Python もカバレッジ測定範囲をカスタマイズできない．Cobertura はパッケージ，クラス，メソッドといった階層による限定は可能であるが，例えば，特定のメソッドを呼び出すステートメントのみに限定するといったことは不可能である．

一方，提案 FW では固有コードを追加することで，測定用コードを埋め込む位置を自由に調整できる．これにより，特定のメソッドを呼び出すステートメントのみを指定して，ステートメントカバレッジを測定するためのコードを埋め込むことができる．このような測定用コードを埋め込んだプログラムを実行すれば，特定のメソッドを呼び出すステートメントのみに限定した特殊なステートメントカバレッジが得られる．なお，MEF によるプラグインアーキテクチャや DLR によるスクリプト言語への対応により，容易に固有コードを追加できる．

以上から，柔軟なカバレッジ測定における問題の解決に成功した．

5.5 完全な測定

デッドコードの測定について議論することで，完全な測定を評価する．

Cobertura は，クラスファイルに測定用コードを埋め込む．しかし，クラスファイルはコンパイラの最適化によってデッドコードが失われており，デッドコードに対して測定用コードを埋め込むことができない．

一方，提案 FW では，コンパイラが最適化を行う前に，ソースコードに測定用コードを埋め込む．そのため，コンパイラの最適化によってデッドコードが失われたとしても，埋め込み時の情報として残り，カバレッジされなかった部分として判断できる．

リスト 7 DeadCode.java

```
1 public class DeadCode {
2   public static void main(String[] args) {
3     System.out.println("main");
4     if (false) {
5       System.out.println("deadcode");
6     }
7   }
8 }
```

リスト 7 に，デッドコードを含むソースコード例を示す．このソースコードを Cobertura でテストすると，ステートメントカバレッジが 100% になるが，提案 FW の実装例で測定すると，ステートメントカバレッジが 50% となる．このように，デッドコードの検出にも役立てることができる．

以上から，コンパイラの影響による不完全な測定の問題の解決に成功した．

5.6 実行効率

カバレッジを測定するためにテストした際の実行時間を比較して，時間効率を評価する．

測定ツールは，テスト時にカバレッジ測定を行うためパフォーマンス低下を引き起こす．ソースコードに測定用コードを埋め込む手法では，埋め込んだコードの実行によりパフォーマンスが低下する．

実際に，Cobertura と提案 FW の実装例とで，書籍から引用した三種類の Java プログラム [17] に対してカバレッジを測定しながらテストを行い，実行時間を測定した．

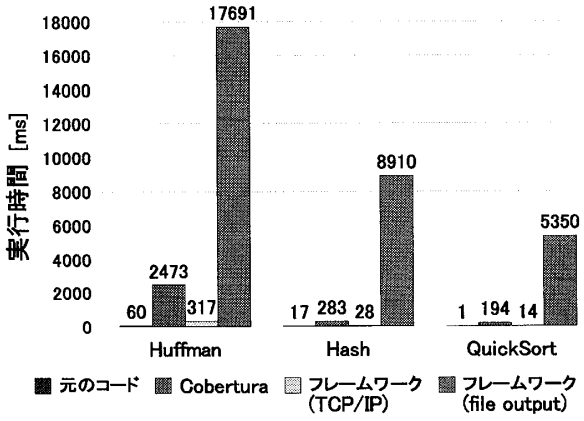

図 10 実行時間

図 10 のように，提案 FW では，同一 PC 内でソケットによる TCP/IP 通信を用いることで，元のソースコードと比較して，2 倍から 10 倍程度の実行時間に抑えており，Cobertura と比較して 10 倍程度早くなっている．ただし，ファイル出力を用いた場合は，元のソースコードと比較して 1000 倍以上，Cobertura と比較して 10 倍から 30 倍程度遅くなっている．そのため，素朴なファイル出力よりも，TCP/IP 通信の方が圧倒的に高速であることが分かる．

以上により，ソケットによる TCP/IP 通信を用いることで，ソフトウェアテストにおける実行効率の低下の問題がないことを確認した．

6. 関連研究

提案 FW の仕組みや目的に関連する研究として，桐井ら [18] の手法と，Hridesh Rajan ら [19] の手法がある．

- 桐井らは，ソースコードに測定用コードを埋め込むことで，測定ツールを開発する手法を提案している．
  桐井らの手法では，カバレッジレベルについて，ステートメントカバレッジとデシジョンカバレッジ以外に，RC0 というレベルに対応している．RC0 とはプログラムを修正した際に，修正した差分のみを測定対象としたステートメントカバレッジのことである．
  しかし，埋め込み方法としては，ステートメントの挿入のみによってカバレッジを測定しており，ステートメントカバレッジとデシジョンカバレッジの二種類のみに対応する．また，Java，C/C++，VisualBasic，ABAP/4 の四種類の言語に対応しているが，それぞれの共通処理などには言及しておらず，新たに対応する言語を増やす際の支援はない．
  一方，提案 FW における実装例では，RC0 に対応していないが，機能拡張と柔軟な測定の支援により，RC0 を含め，新たなカバレッジレベルへの対応が可能である．
- Hridesh Rajan らは，AspectJ などのアスペクト指向言語で使われるポイントカットの記述方法を応用して，測定対象の指定手法を提案している．
  Hridesh Rajan らの手法では，C#言語に対応する測定ツールの実装例が示されている．例えば，メソッド呼び出し，if 文，foreach 文，例外ハンドラ，代入文など様々な測定対象を指定して，測定範囲を限定できる．
  しかし，測定対象を指定する記述モデルは，C#言語に特化しており，C#言語とパラダイムの異なる言語には，記述モデルを利用できない可能性が高い．
  一方，提案 FW では，機能拡張と柔軟な測定により，Hridesh Rajan らの記述モデルを言語非依存に修正した上で，測定対象の指定方法として実装できる．
- その他，三宅ら [20] は複数のプログラミング言語に対応したメトリクス計測プラグインプラットフォームを提案している．三宅らの手法は，AST を介することで複数のプログラミング言語に対応しており，本手法と目的が異なるものの，類似性が見られる．
  また，嶋野ら [21] はシェルスクリプトを XML フォーマットで表現することで，シェルの違いによる文法の差を吸収する手法を提案している．嶋野らの手法は，木構造である XML フォーマットを用いる点で，本手法との類似性が見られる．

7. まとめと展望

本稿では，多言語対応のカバレッジ測定フレームワークを提案した．これにより，共通処理の再利用により開発コストの低減，複数の言語に統一的に対応したカバレッジ測定，機能拡張の支援による柔軟な測定，ソースコードへの埋め込みによる完全な測定をした．

以降，今後の展望について，考えられる改良点を述べる．

1. 対応テストカバレッジの種類の拡大
   マルチプルコンディションカバレッジや桐井らの手法による差分のみを測定対象とするカバレッジなど，様々なカバレッジに対応することが考えられる．
2. カバレッジ測定用コードの実装支援
   測定用コードに対応する AST 上における部分木の生成処理に関して，必要となる部分木そのものを開発者が導き出してから実装する．これに対して，測定用コードの埋め込み前と埋め込み後のコードの差分から，必要となる部分木を提案 FW が導き出すことで，生成処理の実装を半自動化する支援が考えられる．
   また，埋め込んだ測定用コードを AST 上の操作対象にしない場合，測定用コードを部分木ではなく，ただの文字列トークンとして埋め込んでも良い．そのため，測定用コードに対する AST 上の操作を想定しなければ，部分木を生成する処理の実装を大幅に簡略化できる．
3. カバレッジ測定用コード本体の実装支援
   測定用コードが呼び出すサブルーチン本体の実装への支援が必要である．C/C++ で記述されたプログラムやライブラリを様々な言語から呼び出せるように，インタフェース部分のコードを自動生成する SWIG (Simplified Wrapper and Interface Generator)[22] というソフトウェアがある．これを利用することで，サブルーチン本体の実装の手間を省くことができる．
4. カバレッジ表示サブシステムの改良
   Java 言語におけるパッケージ，クラス，メソッドといった階層情報をタグ情報に含めて，測定結果を階層情報でフィルタリングする改良が考えられる．
   また，プロファイリングツールとしての応用が考えられる．測定対象要素の実行回数を取得できるため，例えば，各ステートメントや条件分岐における各分岐先の実行回数を表示できる．

参考文献

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

[2] 坂本一憲, Open Code Coverage Framework, http://sourceforge.jp/projects/codecoverage/.

[3] Sun Microsystems, J2SE(TM) 5.0 New Features, http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html.

[4] the GNU Compiler Collection, http://gcc.gnu.org/.

[5] Yuji Sakata, Kazutoshi Yokoyama, Hironori Washizaki and Yoshiaki Fukazawa, "A precise estimation technique for test coverage of components in object-oriented frameworks", 13th Asia-Pacific Software Engineering Conference (APSEC '06), IEEE CS, pp.79-86, 2006.

[6] Cobertura, http://cobertura.sourceforge.net/.

[7] 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.

[8] Microsoft, .NET Framework, http://msdn.microsoft.com/netframework/.

[9] Microsoft, Dynamic Language Runtime, http://dlr.codeplex.com/.

[10] Microsoft, Managed Extensibility Framework, http://www.codeplex.com/MEF/.

[11] Etienne M. Gagnon, Ben Menking, Mariusz Nowostawski, Komivi Kevin Agbakpem and Kis Gergely, SableCC, http://sablecc.org/.

[12] ANTLR, http://www.antlr.org/.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.

[14] OMG, Unified Modeling Language (UML) specification, version 2.2, http://www.omg.org/spec/UML/.

[15] Gareth Rees, Statement coverage for Python, http://garethrees.org/2001/12/04/python-coverage/.

[16] Apache Software Foundation, The Byte Code Engineering Library, http://jakarta.apache.org/bcel/.

[17] 奥村晴彦, 杉浦方紀, 津留和生, 首藤一幸, 土村展之著, "Java によるアルゴリズム事典,", 技術評論社, 2003.

[18] 桐井隆志, 三好辰弥, 岸上諭, 大里立夫, 曽根原勝, "ソースコード埋め込み型カバレッジツールについて", 情報処理学会第 65 回全国大会, 2003.

[19] Hridesh Rajan and Kevin Sullivan, "Aspect Language Features for Concern Coverage Profiling", International Conference on Aspect-Oriented Software Development (AOSD'05), pp181-191, 2005.

[20] 三宅達也, 肥後芳樹, 井上克郎, "メトリクス計測プラグインプラットフォーム MASU の開発", ソフトウェアエンジニアリングシンポジウム 2008, pp63-70, 2008.

[21] 嶋野淳子, 新家博文, 四野見秀明, "シェルスクリプトのための XML モデル", 情報処理学会 ソフトウェア工学研究会, Vol.2008, No.29, pp211-218, 2008.

[22] Simplified Wrapper and Interface Generator, http://www.swig.org/.