OCCF: A Framework for Developing Test Coverage Measurement Tools Supporting Multiple Programming Languages

Kazunori Sakamoto, Kiyofumi Shimojo, Ryohei Takasawa, Hironori Washizaki, Yoshiaki Fukazawa
Dept. Computer Science and Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku
Tokyo 1698555, Japan
{kazuu@ruri, kiyofumi-1906@akane, ryohei-takasawa@asagi} .waseda.jp, {washizaki, fukazawa} @waseda.jp

***Abstract*—Although many programming languages and test coverage criteria currently exist, most coverage measurement tools only support select programming languages and coverage criteria. Consequently, multiple measurement tools must be combined to measure coverage for software which uses multiple programming languages such as web applications. However, such combination leads to inconsistent and inaccurate measurement results.**

In this paper, we describe a consistent and flexible framework for measuring coverage supporting multiple programming languages, called Open Code Coverage Framework (OCCF). OCCF allows users to add new extensions for supporting programming languages and coverage criteria with low development costs. To evaluate the effectiveness of OCCF, sample implementation to support statement coverage and decision coverage for eight programming languages (C, C++, C#, Java, JavaScript, Python, Ruby and Lua) are demonstrated. Additionally, applications of OCCF for localizing faults and minimizing tests are shown.

***Index Terms*—framework; test coverage; programming languages; fault localization; test-suite minimization;**

I. Introduction

Test coverage, which is an important indicator for test adequacy [1], has various criteria such as statement coverage, decision coverage, condition coverage and condition/decision coverage. For example, statement coverage is the ratio of statements that have been executed at least once from all the statements. Based on the purpose of software testing, the developer selects the suitable criterion [2].

It is more difficult to measure coverage than software metrics, such as McCabe complexity [3] and CK metrics [4], because any three mechanisms to measure coverage (extending interpreters or processors, inserting instrumentation code into binary code, or inserting instrumentation code into source code) are complex. In general, extending interpreters or processors is more difficult than inserting instrumentation code because interpreters or processors are such complex systems that their extensions require significant efforts. Two insertion mechanisms must analyze and transform source code or binary code, while measuring software metrics must only analyze source code.

Although many programming languages and coverage criteria exist, most coverage-measurement tools support select programming languages and coverage criteria. Consequently, many tools exist to span various programming languages, which leads to differences between existing tools [5]. These differences prevent testers from accurately measuring coverage because the tools support different coverage criteria and are implemented with different ways, resulting in different values for the same criteria. Such a problem is also recognized for the software metrics to analyze solely the source code [6].

For example, EMMA supports statement coverage for Java, while Coverage.py supports both statement coverage and decision coverage for Python. Although a combination of EMMA and Coverage.py can measure statement coverage of a web application using Java and Python, the combination cannot measure decision coverage. Moreover, EMMA divides a ternary expression (`condition ? true-expression : false-expression`) into two statements and can determine whether both branches of the ternary expression have been executed. On the other hand, Coverage.py cannot determine whether both branches have been executed because it does not divide ternary expressions.

To overcome the diversity of existing tools, we developed a novel framework for consistently and flexibly measuring the coverage supporting multiple programming languages, called Open Code Coverage Framework (OCCF) [7], [8]. OCCF reduces the development costs of coverage-based tools by providing reusable code as cold spots. OCCF provides common features for all supported programming languages to measure the four coverage criteria: statement coverage, decision coverage, condition coverage and decision/condition coverage. OCCF allows users to add extensions for supporting new programming languages and coverage criteria.

In this paper, we describe the features and architectures of OCCF. We also demonstrate code fragments to measure statement coverage and decision coverage for eight programming languages. Moreover, code fragments for reporting coverage, localizing faults, and minimizing test cases based on coverage are explained to evaluate the effectiveness of OCCF.

TABLE I
Comparison of coverage-measurement tools for features and coverage criteria

Tool	Instrumentation	Minimal targets	Free	Coverage criteria			
				Statement	Decision	Condition	Condition/Decision
Cobertura	Binary	Function	X	X	X		
EMMA	Binary	Function	X	X	X		
JCover	Code	Function		X	X		
Clover	Code	Function		X	X		
Agitar	Binary	Function		X	X		
OpenCover	Processor	Function	X	X	X		
NCover	Source	Function		X	X		
dotCover	Binary	Function		X			
gcov	Binary(Compiler)	File	X	X	X		
COVTOOL	Code	File	X	X			
BullseyeCoverage	Code	Function		X	X	X	X
Intel Code Coverage Tool	Binary(Compiler)	Function		X			
Squish Coco	Code	Function		X	X	X	X
TCAT	Code	Function			X		
Parasoft Test	Code	Function		X	X	X	X
PurifyPlus	Binary	Function		X	X	X	X
Semantic Designs	Code	Function		X	X		
CoverageValidator	Code	Function		X	X		
ScriptCover	Code	File	X	X			
undercover	Binary	Function	X	X	X		
Coverage.py	Processor	Module	X	X	X		
rcov	Processor	Function	X	X			
SimpleCov	Processor	Function	X	X			
Devel::Cover	Processor	Function	X	X	X	X	
xdebug	Code	Function	X	X			
LuaCov	Processor	File	X	X			

TABLE II
Comparison of coverage-measurement tools for supported programming languages

Tool	Languages										
	C/C++	C#	Java	Scala	Groovy	JavaScript	Python	Ruby	Perl	PHP	Lua
Cobertura			X								
EMMA			X								
JCover			X								
Clover			X		X						
Agitar			X								
OpenCover		X									
NCover		X									
dotCover		X									
gcov	X										
COVTOOL	X										
BullseyeCoverage	X										
Intel Code Coverage Tool	X										
Squish Coco	X										
TCAT	X		X								
Parasoft Test	X	X	X								
PurifyPlus	X	X	X								
Semantic Designs	X	X	X							X	
CoverageValidator	X	X	X								
ScriptCover						X					
undercover			X	X							
Coverage.py							X				
rcov								X			
SimpleCov								X			
Devel::Cover									X		
xdebug										X	
LuaCov											X

TABLE III
Relation between user code, common code and external program for each module

Module	User code (hot spot)	Common code (clod spot)	External program
AST generator	Passing a file path of a parser and arguments for the parser	Invoking the parser	A parser library or a compiler compiler
AST finder	Finding AST nodes where instrumentation code is inserted	Scanning ASTs	–
AST inserter	Passing the string of instrumentation code	Inserting new nodes into ASTs	–
Instrumentation code	Implementing instrumentation code in each language	–	SWIG
Code generator	Outputting special tokens not memorized in AST	Outputting memorized tokens	–

II. Comparison to existing coverage tools

Tables I and II compare existing coverage-measurement tools by features and supported programming languages. An 'X' in the tables indicates the tool has the feature to support a criterion or programming language. To the best of our knowledge, no free tool supports the four coverage criteria. Few tools support the scripting languages such as JavaScript, Python, Ruby and Lua. Consequently, these languages have poor features and support few coverage criteria. Although undercover is a free tool, which supports more than one programming language, undercover targets only programming languages working on Java Virtual Machine (Java VM). Non-free tools, except Clover and Semantic Designs, support only C/C++, C# and Java because these programming languages are most common. Therefore, accurately measuring coverage for web applications using scripting languages is difficult.

A new mechanism to reuse common code between various programming languages beyond platforms such as Java VM and .NET Framework must be developed to provide appropriate tools with rich features that support many programming languages. However, programming languages have different grammars and features. In particular, compilers and processors are developed in various ways. Thus, extending compilers or processors is a very language-dependent approach.

III. Overview of OCCF

To alleviate the current problematic state, we developed OCCF, which inserts instrumentation code into source code. The abstract syntax trees (ASTs) of source code for most programming languages have similar structures. Thus, OCCF provides a reusable common code to insert instrumentation code through ASTs by utilizing the similarities. Currently, we have developed parsers for C, C++, C#, Java, JavaScript, Python, Ruby and Lua.

A. Approach for measuring test coverage

OCCF inserts instrumentation code into the source code through ASTs. Coverage is measured by executing the program after the insertion. Our approach generates the ASTs from the source code and locates the nodes where the instrumentation code is inserted by analyzing the syntax and the required part of the semantics. Our approach has commonality among the measurement features because the insertion process through the ASTs is similar for each programming language.

Lists 1 and 2 outline the source code written in C prior to and after inserting the instrumentation code, respectively. The `stmt_record` and `decision_record` functions save which items are executed with the identities of the items. The `decision_record` function returns the evaluation value of the original conditional expression. OCCF inserts the `stmt_record` into each statement and variable initializer to measure the statement coverage. In addition, OCCF inserts the `decision_record` into each conditional expression of the control-flow statement and the `stmt_record` into each case clause of the switch statement to measure decision coverage, condition coverage and condition/decision coverage.

```c
int main() {
  int a = 0;
  printf("test");
  if (a == 0) { puts("a == 0"); }
  else {        puts("a != 0"); }
}
```

List 1. Sample code written in C prior to inserting the instrumentation code

```c
int main() {
  int a = stmt_record(0) ? 0 : 0;
  stmt_record(1); printf("test");
  if (decision_record(0, a == 0)) {
    stmt_record(2); puts("a == 0");
  }
  else {
    stmt_record(3); puts("a != 0");
  }
}
```

List 2. Sample code written in C after inserting the instrumentation code

The potential drawback to the instrumentation code is changing timing behaviour by processing to collect coverage information. However, this drawback exist not only in our approach but also other approaches because saving execution logs requires a computational cost. Moreover, the execution environment always affect the timing behaviour. Consequently, this drawback might be ignored.

B. Approach for absorbing the differences between programming languages

Lists 3, 4 and 5 show the fragments of ANTLR BNF grammars for C, Java and Python, respectively. To measure decision coverage, the instrumentation code should be inserted into the conditions of `if` statements. `if` statements belong to `statement` or `compound_stmt` elements. The conditions of the three programming languages are the `expression` element in the `selection_statement` element for C, the `statement` element in the `parExpression` element for Java, and the `if_stmt` element in the `test` element for Python. In this way, these programming languages have similar elements with different names. Consequently, OCCF absorbs the differences between programming languages by adding element names for each programming languages as hot spots.

```
statement
  : labeled_statement
  | compound_statement
  | expression_statement
  | selection_statement | ... ;

selection_statement
  : 'if' '(' expression ')' statement ('else' statement)?
  | 'switch' '(' expression ')' statement ;
```

List 3. Fragment of BNF grammar related to an `if` statement for C

```
statement
  : 'if' parExpression statement ('else' statement)?
  | expression ';' | ... ;

parExpression
  : '(' expression ')' ;
```

List 4. Fragment of BNF grammar related to an `if` statement for Java

```
compound_stmt
  : if_stmt | while_stmt | for_stmt | try_stmt
  | with_stmt | funcdef | classdef | decorated

if_stmt
  : 'if' test ':' suite ('elif' test ':' suite)*
    ('else' ':' suite)? ;
```

List 5. Fragment of BNF grammar related to an `if` statement for Python

C. Architecture of OCCF

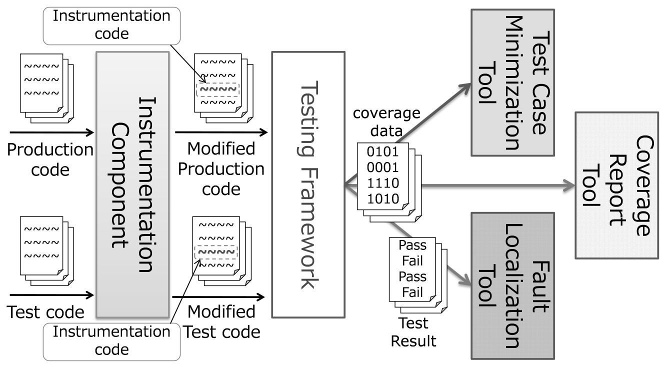


Fig. 1. Overview and measurement process of OCCF

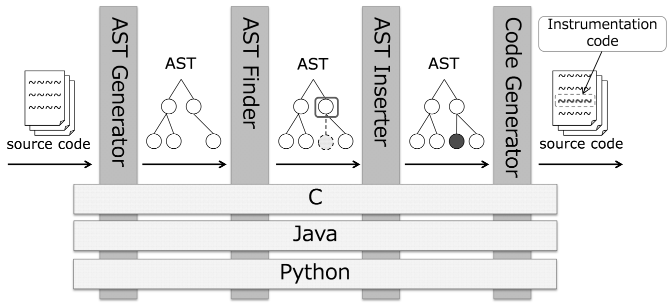


Fig. 2. Architecture of the instrumentation component in OCCF

Figure 1 overviews the measurement process of OCCF. In addition to the core component, called the instrumentation component, OCCF has three tools: the coverage-report, fault-localization and test-suite minimization tools. The instrumentation component consists of four modules: the AST generator, AST finder, AST inserter and code generator as shown in Figure 2.

OCCF adopts the general architecture of a coverage-measurement tool that employs the insertion approach of the instrumentation code into source code. Adding the four modules for each programming language allows developers to add new extensions using their preferred parsers because OCCF does not require a language-independent AST.

The process to measure coverage has the following six steps. 1) The AST generator produces the ASTs from the given source code. 2) The AST finder locates the AST nodes where the instrumentation code is inserted. 3) The AST inserter adds the instrumentation code into the ASTs. 4) The code generator produces the source code from the ASTs. 5) A testing framework executes test cases with the generated source code and outputs the measurement information. 6) The coverage-report tool displays the measurement results.

OCCF saves the pair of identifiers which are assigned with each inserted node and the information about its location on source code to a file named ".occf_coverage_inf". The instrumentation code passes execution traces to the coverage-report tool via a file or a shared memory. The coverage-report tool judges which program elements are executed or not by analyzing ".occf_coverage_inf" and execution traces from a file or a shared memory. Although shared memory accesses are faster than file accesses, the instrumentation code must be executed when the coverage-report tool are running.

OCCF provides common code for language-independent processing that manipulates the ASTs using similar structures and provides a design to aid in writing user code for language-dependent processing. Table III lists the relations between the common code, user code and external program. Although the architecture of OCCF reduces the development and maintenance costs, OCCF only targets procedural programming languages and impure functional programming languages owing to its insertion approach.

IV. Implementation of OCCF

OCCF is developed with C# 4.0 and runs on .NET Framework and Mono. OCCF utilizes LINQ to XML [9] to simplify the OCCF source code. LINQ to XML is a powerful library based on LINQ technology to manipulate XML, and provides scanning methods that return lazy lists of various XML elements such as descendant elements. OCCF is published as a NuGet package [1]. Additionally, a set of parsers for the AST generators and the code generators, called Code2Xml [10], is published separately as another NuGet package [2].

Figure 3 shows a class diagram of core classes in OCCF. The `Occf.Core` namespace indicates the instrumentation component. The `Occf.Core.CoverageInformation` and `Occf.Core.TestInformation` namespaces contain the entity classes related to coverage information about measurement targets such as statements and test information about test cases, respectively. The `Occf.Core.Tools.Cui` namespace contains the executable programs as tools. The `Occf.Core` namespace depends on the `Occf.Core.CoverageInformation` `Occf.Core.TestInformation` namespaces, while the `Occf.Core.Tools.Cui` namespace depends on the other namespaces because the tools are developed with OCCF.

The classes in Figure 3 are common code provided by OCCF. To add extensions for supporting new programming language, developers can extend the `LanguageSupport` abstract class, which depends on the `AstFinder` and `AstInserter` abstract classes. OCCF automatically loads the classes that extend the `LanguageSupport` abstract class

[1]https://nuget.org/packages/OpenCodeCoverageFramework
[2]https://nuget.org/packages/Code2Xml

because these abstract classes should be extended for each programming language. Developers can write code reusing the classes in the `Occf.Core` namespace to develop new coverage-based tools.

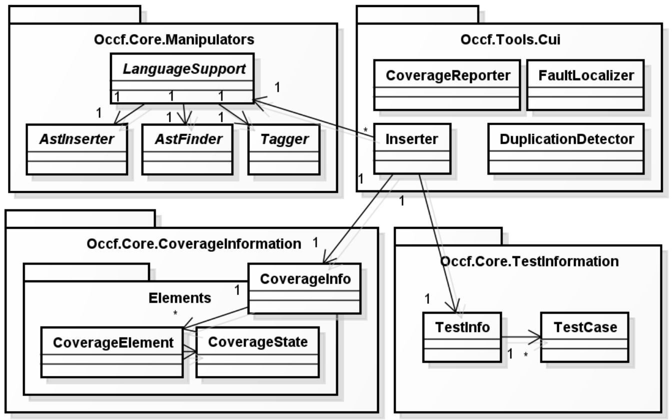


Fig. 3. Class diagram of the core classes in OCCF

V. Implementation of extensions for supporting eight programming languages

Below we describe code fragments for methods to find statements and `if` conditions for C, C++, C#, Java, JavaScript, Python, Ruby and Lua. The method extracts statement elements, except for blocks, labeled statements and empty statements because the exceptions do nothing when executing and the child statements in their elements are also extracted. Although more elements must be extracted to measure decision coverage, for simplicity, our methods only extract conditions of `if` statements. Developers can customize coverage criteria by changing these methods (e.g., filtering specific statements).

We use a compiler compiler and existing parser libraries to implement parsers in the Code2Xml package We use ANTLR [11], which is a compiler compiler to implement parsers for C, Java, JavaScript and Lua. We also use three parser libraries: srcML [12] for C++, the parser module in the Python standard library for Python and ruby_parser [13] for Ruby.

Our parsers with ANTLR memorize tokens of the original source code in the ASTs to easily convert the ASTs into source code. The `AntlrCodeToXml` abstract class is used to implement the code generator. Although we use existing parser libraries for C++, Python and Ruby instead of ANTLR, these parser libraries also provide mechanisms to convert the ASTs into source code, allowing developers to focus on implementing the AST finder and the AST inserter, including instrumentation code for each programming language. The following code fragments are very simple and similar because the grammars for programming languages have similar structures.

A. Methods to find statements and if conditions for C

OCCF uses ANTLR and the BNF grammar [3] to implement a C parser. Lists 6 and 7 show code fragments in OCCF for C.

[3]/Code2Xml.Languages/C/CodeToXmls/C.g on our GitHub repos

The `statement` element consists of more detailed statement elements such as `labeled_statement` and `compound_statement`. Because `labeled_statement` and `compound_statement` have `statement` as child elements, this `FindStatements` method excludes these elements to avoid redundantly inserting instrumentation code. The `selection_statement` element indicates `if` and switch-case statements. This `FindBranches` method extracts only the `if` statements with the `Where` methods.

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("statement")
      .Select(e => e.FirstElement())
      .Where(e => e.Name() != "labeled_statement")
      .Where(e => e.Name() != "compound_statement");
}
```

List 6. Method to find statements for C

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("selection_statement")
      .Where(e => e.FirstElement().Value == "if")
      .Select(e => e.NthElement(2));
}
```

List 7. Method to find if conditions for C

B. Methods to find statements and if conditions for C++

OCCF uses SrcML to implement the C++ parser. Lists 8 and 9 show code fragments in OCCF for C++.

This `FindStatements` method extracts all statements except for blocks, labels and empty statements. The C++ parser names blocks, labeled statements and empty statements, which makes the `FindStatements` method very simple. This `FindBranches` is simpler than the method for C because the `if` element indicates just `if` statements.

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("block")
      .SelectMany(e => e.Elements())
      .Where(e => e.Name() != "block")
      .Where(e => e.Name() != "label")
      .Where(e => e.Name() != "empty_stmt");
}
```

List 8. Method to find statements for C++

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("if")
      .Select(e => e.Element("condition"));
}
```

List 9. Method to find if conditions for C++

C. Methods to find statements and if conditions for C#

OCCF uses ANTLR and the BNF grammar [4] to implement a C# parser.

Lists 10 and 11 show code fragments in OCCF for C#. This `FindStatements` method extracts statements, including `declaration_statement`, because C# allows a mixture of variable declarations and other statements. This `FindBranches` is similar to the method for C++.

[4]/Code2Xml.Languages/CSharp/CodeToXmls/cs.g on our GitHub repos

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  var decls = root.Descendants("declaration_statement");
  var stmts = root.Descendants("embedded_statement")
      .Where(e => e.FirstElement().Name() != "block");
  return stmts.Concat(decls);
}
```

List 10. Method to find statements for C#

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("if_statement")
      .Select(e => e.Element("boolean_expression"));
}
```

List 11. Method to find if conditions for C#

D. Methods to find statements and if conditions for Java

OCCF uses ANTLR and the BNF grammar [5] to implement the Java parser. Lists 12 and 13 show code fragments in OCCF for Java.

This `FindStatements` method excludes labeled statements that use the ':' character because this grammar does not give a name to the labeled statement. This `FindBranches` method digs the `parExpression` element because this element contains '(' and ')' characters.

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("statement")
      .Where(e => e.FirstElement().Name() != "block")
      .Where(e => e.NthElementOrDefault(1)
           .SafeValue() != ":")
      .Where(e => e.FirstElement().Value != ";");
}
```

List 12. Method to find statements for Java

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("statement")
      .Where(e => e.FirstElement().Value == "if")
      .Select(e => e.Element("parExpression"))
      .Select(e => e.NthElement(1));
}
```

List 13. Method to find if conditions for Java

E. Methods to find statements and if conditions for JavaScript

OCCF uses ANTLR and the BNF grammar [6] to implement the JavaScript parser. Lists 14 and 15 show code fragments in OCCF for JavaScript. Similar to the method for C++, this `FindStatements` and `FindBranches` methods are very simple.

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("statement")
      .Select(e => e.FirstElement())
      .Where(e => e.Name() != "statementBlock")
      .Where(e => e.Name() != "labeledStatement")
      .Where(e => e.Name() != "emptyStatement");
}
```

List 14. Method to find statements for JavaScript

[5]/Code2Xml.Languages/Java/CodeToXmls/Java.g on our GitHub repos

[6]/Code2Xml.Languages/JavaScript/CodeToXmls/JavaScript.g on our GitHub repos

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("ifStatement")
      .Select(e => e.Element("expression"));
}
```

List 15. Method to find if conditions for JavaScript

F. Methods to find statements and if conditions for Python

OCCF uses the Python standard library to implement the Python parser. Lists 16 and 17 show code fragments in OCCF for Python. This parser distinguishes between statements with a block called `compound_stmt`, and statements without a block, called `small_stmt`.

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("compound_stmt")
      .Concat(root.Descendants("small_stmt"))
}
```

List 16. Method to find statements for Python

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("if_stmt")
      .SelectMany(e => e.Elements("test"));
}
```

List 17. Method to find if conditions for Python

G. Methods to find statements and if conditions for Ruby

OCCF uses the ruby_parser. Lists 18 and 19 show code fragments in OCCF for Ruby. This `FindStatements` method only excludes blocks because Ruby does not have labeled statements. This parser also removes empty statements.

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("block")
      .SelectMany(e => e.Elements())
      .Where(e => e.Name() != "block");
}
```

List 18. Method to find statements for Ruby

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("if")
      .Select(e => e.FirstElement());
}
```

List 19. Method to find if conditions for Ruby

H. Methods to find statements and if conditions for Lua

OCCF uses ANTLR and the BNF grammar [7] to implement the Lua parser. Lists 20 and 21 show code fragments for Lua. This `FindStatements` method extracts `stat` and `laststat` elements because this grammar distinguishes between statements and last statements in functions such as `return` and `break`. Lua does not allow empty statements such as those which have only ';'.

[7]/Code2Xml.Languages/Lua/CodeToXmls/Lua.g on our GitHub repos

```
public override IEnumerable<XElement>
    FindStatements(XElement root) {
  return root.Descendants("stat")
      .Concat(root.Descendants("laststat"))
}
```

List 20. Method to find statements for Lua

```
public override IEnumerable<XElement>
    FindBranches(XElement root) {
  return root.Descendants("stat")
      .Where(e => e.FirstElement().Value == "if")
      .Select(e => e.NthElement(1));
}
```

List 21. Method to find if conditions for Lua

VI. Implementation of coverage-based tools

In this section, we show code fragments of three tools: the coverage-report, fault-localization and test-suite minimization tools. These tools help develop coverage-based tools.

A. Coverage-report tool

The coverage-report tool reads coverage information and coverage data. The coverage information indicates what statements exist in the specific source code by inserting the instrumentation code into the source code. The coverage data, which is generated when executing test cases, indicates which statements are executed in testing.

List 22 shows the code fragment of the tool. `covInfo` determines the measurement results by merging coverage information and coverage data. `nExe` and `nAll` are variables denoting the number of executed statements and all statements, respectively.

```
var executedAndNot = covInfo.StatementTargets
  .Where(e => e.Tag.StartsWith(tag))
  .Halve(e => e.State == CoverageState.Done);
var nExe = executedAndNot.Item1.Count;
var nAll = nExe + executedAndNot.Item2.Count;
Console.WriteLine("Statement Coverage: "
  + nExe * (100.0 / nAll) + "% : " + nExe + " / " + nAll);
Console.WriteLine("Not executed statements:");
foreach (var element in executedAndNot.Item2)
  Console.Write(element.Position.SmartPositionString);
```

List 22. Code fragment to report coverage

B. Fault-localization tool

The fault-localization tool reads coverage information, test information, coverage data and test results. The test information indicates which test cases execute which statements, and it is generated when the instrumentation code is inserted into the source code. The test results indicate whether the test case passed or failed. The tool calculates and reports the risk values for each statement with the formula proposed by Jones et al. [14]. Note that the tool allows users to change the formula by modifying the IronPython script file.

List 23 shows the code fragment of the tool. Merging test information and test results provides `testInfo`. `passedTestCases` and `executedAndPassedTestCases` are variables that indicate a set of passed test cases and a set of passed and executed test cases, respectively. In contrast, `failedTestCases` and `executedAndFailedTestCases` are variables that indicate a set of failed test cases and a set of failed and executed test cases, respectively. `CalculateRisk` is a method to calculate risk values using the formula defined in the IronPython script file.

```
foreach (var stmt in covInfo.StatementIndexAndTargets) {
  var index = stmt.Item1;
  var stmtInfo = stmt.Item2;
  var passedTestCases = testInfo.TestCases
      .Where(t => t.Passed);
  var executedAndPassedTestCases = passedTestCases
      .Where(t => t.Statements.Contains(index));
  var failedTestCases = testInfo.TestCases
      .Where(t => !t.Passed);
  var executedAndFailedTestCases = failedTestCases
      .Where(t => t.Statements.Contains(index));

  var executedAndPassedCount =
      executedAndPassedTestCases.Count();
  var passedCount = passedTestCases.Count();
  var executedAndFailedCount =
      executedAndFailedTestCases.Count();
  var failedCount = failedTestCases.Count();
  var risk = CalculateRisk(
      executedAndPassedCount, passedCount,
      executedAndFailedCount, failedCount);
  Console.WriteLine(stmtInfo.Position + ": " + risk);
}
```

List 23. Code fragment to localize fault based on coverage

C. Test-suite minimization tool

The test-suite minimization tool reads coverage information and coverage data. The tool calculates a set of statements executed for each test case. To detect duplicated test cases, the tool judges whether a set of statements executed by a test case is a subset of statements executed by another test case.

```
Func<TestCase, TestCase, bool> isDuplicated = (tc, tc2) =>
    tc.Statements.IsSubsetOf(tc2.Statements);
foreach (var tc in testInfo.TestCases) {
  var dups = testInfo.TestCases
      .Where(tc2 => tc2 != tc && isDuplicated(tc, tc2));
  Console.WriteLine(tc.Name + " is duplicated with:");
  foreach (var dup in dups) {
    Console.WriteLine(dup.Name);
  }
}
```

List 24. Code fragment to minimize test cases based on coverage

List 24 shows the code fragment of the tool. `isDuplicated` is a function object to judge such duplications. The tool allows users to change the judging criterion by utilizing other coverage criteria.

VII. Related work

Qian et al. [15] surveyed coverage-based tools, and found 17 tools to compare three features: (1) code coverage measurement, (2) coverage criteria and (3) automation and reporting. Section 2 showed 26 tools based on their results with an emphasis on supported programming languages and coverage criteria. Although they did not show the problem with current coverage-measurement tools, we show it by comparing existing tools. Muhammad et al. [16] also surveyed 47 research papers related to test coverage and reported three frameworks including OCCF to measure test coverage. They found that only OCCF assisted in adding extensions for supporting new programming languages.

Higo et al. [17] proposed a framework for measuring software metrics supporting multiple programming languages, called MASU. MASU constructs language-independent ASTs, which contain necessary information to measure software metrics such as CK metrics. Unlike OCCF, which measures coverage by analyzing and transforming source code, MASU just analyzes source code. Moreover, it is easier to add extensions for supporting new programming language with OCCF because OCCF uses different ASTs for each programming language.

Rajan et al. proposed the idea of specifying the measuring elements using the description style of pointcut, which is used in Aspect-oriented programming languages [18]. They demonstrated their tool that supported C#. Their description style can specify measurement elements such as method calls, `if` statements, exception handlers and variables. Because the description style was specialized for C#, it cannot be used for programming languages with different paradigms. However, OCCF can measure various coverage including such coverage by adding user code to extract elements to be measured.

VIII. Limitations

Supportable programming languages: The approach of inserting instrumentation code into source code cannot be applied to several non-procedure-oriented languages. When OCCF cannot insert instrumentation code into a location where the instrumentation code is executed just before executing target program elements, OCCF cannot support such programming languages. Moreover, OCCF cannot support programming languages which have no feature to save execution traces (e.g. whitespace). However, OCCF can support major programming languages because most of programming languages is not under these constraints.

Measurement environment: OCCF requires source code to measure test coverage. In particular, source code where instrumentation code is inserted must be compiled and executed to measure test coverage. Moreover, OCCF is implemented using .NET Framework such that users of OCCF and developers of extensions for OCCF must use a .NET environment such as .NET Framework and Mono. However, test coverage is usually utilized in white-box testing, and thus, testers can easily acquire source code and the development environments. Moreover, we can freely install .NET Framework and Mono on Windows, Mac OS and Linux.

IX. Conclusion

We proposed OCCF to reduce the development costs by extracting the commonalities from multiple programming languages using ASTs. We showed the implementation of extensions for supporting eight programming languages: C, C++, C#, Java, JavaScript, Ruby, Python and Lua as well as the implementation of three coverage-based tools to confirm the effectiveness of OCCF.

In the future, we plan to improve OCCF to support non-procedure-oriented languages such as impure functional programming languages. Moreover, we intend to semi-automatically generate all the components using a wizard and the required user input through the GUI to further reduce the developmental costs.

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[2] L. Copeland, *A Practitioner's Guide to Software Test Design*. Norwood, MA, USA: Artech House, Inc., 2003.

[3] T. J. McCabe, "A complexity measure," in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 407–.

[4] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '91. New York, NY, USA: ACM, 1991, pp. 197–211.

[5] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *Proceedings of the 2006 international workshop on Automation of software test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 99–103.

[6] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 131–142.

[7] K. Sakamoto, H. Washizaki, and Y. Fukazawa, "Open code coverage framework: A consistent and flexible framework for measuring test coverage supporting multiple programming languages," in *Proceedings of the 2010 10th International Conference on Quality Software*, ser. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 262–269.

[8] K. Sakamoto, F. Ishikawa, H. Washizaki, and Y. Fukazawa, "Open code coverage framework: A framework for consistent, flexible and complete measurement of test coverage supporting multiple programming languages." *IEICE Transactions*, vol. 94-D, no. 12, pp. 2418–2430, 2011.

[9] Microsoft. Linq (language-integrated query). [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx

[10] K. Sakamoto. Code2xml. [Online]. Available: https://github.com/exKAZUu/Code2Xml/

[11] T. J. Parr and R. W. Quong, "Antlr: a predicated-ll(k) parser generator," *Softw. Pract. Exper.*, vol. 25, pp. 789–810, July 1995.

[12] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight transformation and fact extraction with the srcml toolkit," in *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 173–184.

[13] E. H. Ryan Davis. Rubyforge: Parsetree - ruby parse tree tools: Project info. [Online]. Available: http://rubyforge.org/projects/parsetree/

[14] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[15] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *Comput. J.*, vol. 52, no. 5, pp. 589–597, 2009.

[16] S. Muhammad, I. Suhaimi, and N. M. Mohd, "A study on test coverage in software testing," in *Proceedings of the International Conference on Telecommunication Technology and Applications*, 2011, pp. 207–215.

[17] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, "A pluggable tool for measuring software metrics from source code," in *Proceedings of the 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, ser. IWSM-MENSURA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 3–12.

[18] H. Rajan and K. Sullivan, "Aspect language features for concern coverage profiling," in *Proceedings of the 4th international conference on Aspect-oriented software development*, ser. AOSD '05. New York, NY, USA: ACM, 2005, pp. 181–191.

APPENDIX
PLAN FOR OUR LIVE DEMONSTRATION

In this appendix, we describe our plan for a live demonstration. Although our ultimate goal is advance testing tool technologies, we have four objectives for this live demonstration.

- To discuss the problems of current coverage-measurement tools
- To encourage framework or tool developers to employ OCCF
- To aid testing-tool and programming language developers by explaining how to implement extensions for supporting programming-language with OCCF
- To explain how to develop coverage-based tools using OCCF

We have already successfully developed three testing tools with OCCF. This demonstration of OCCF should help testing tool developers realize lost cost development of new coverage-based tools. Below is an explanation of how we will achieve each objective.

A. Problems of current coverage measurement tools

We will compare existing coverage-based tools and discuss issues with the current state by highlighting the following:

- Free tools support less coverage criteria than non-free tools.
- Non-free tools support only popular programming languages.
- Few or no tools support scripting languages and new programming languages.

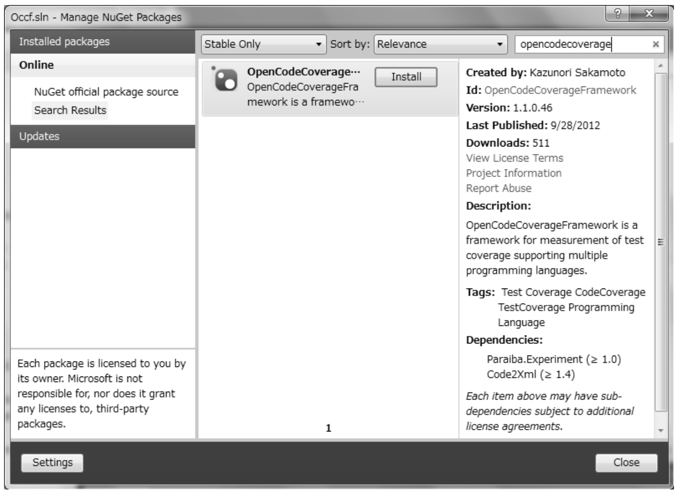
Additionally, we will demonstrate how the current situation causes problems with concrete examples.

B. Architecture and design of OCCF

We will show the architecture and design of OCCF as well as discuss the maintainability and expandability to add extensions for supporting new programming-language and to develop new coverage-based tools. In particular, we will explain the loose coupling of each component. We will show that we published NuGet packages to release OCCF as a set of separate components and show how to import OCCF or partial components in a project by showing Visual Studio with the plug-in of NuGet Package Manager, as shown in Figure 4.

C. Implementation of exntensions for supporting programming-language in OCCF

To aid in the understanding of our measurement approach, which inserts instrumentation code, we will implement extensions for several programming languages. We will explain the similarities between the grammar structures of most programming languages by comparing the grammars and select subtress of ASTs. Then we will demonstrate how utilizing these similarities to reduce the development costs to support multiple programming languages. Finally, we will write sample code to support some programming languages in OCCF using Visual Studio.



OpenCodeCoverage…
OpenCodeCoverageFramework is a framewo…

Created by: Kazunori Sakamoto
Id: OpenCodeCoverageFramework
Version: 1.1.0.46
Last Published: 9/28/2012
Downloads: 511
View License Terms
Project Information
Report Abuse
Description:
OpenCodeCoverageFramework is a framework for measurement of test coverage supporting multiple programming languages.
Tags: Test Coverage CodeCoverage TestCoverage Programming Language
Dependencies:
Paraiba.Experiment (≥ 1.0)
Code2Xml (≥ 1.4)
Each item above may have sub-dependencies subject to additional license agreements.

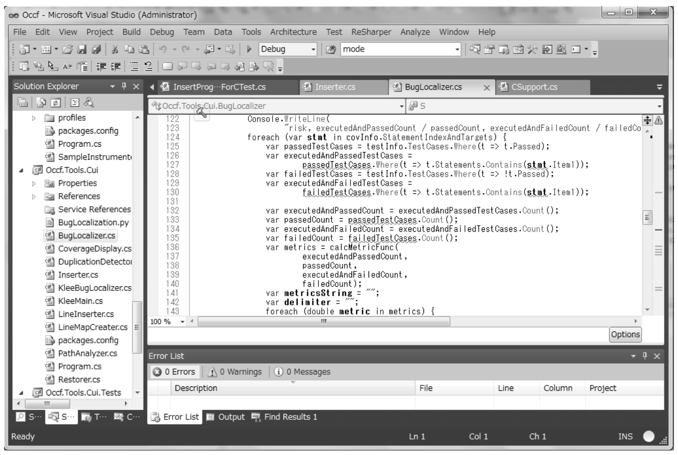
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Fig. 4. Screenshot when importing OCCF with NuGet Package Manager

D. Implementation of coverage-based tools using OCCF

To demonstrate that OCCF works well to support different programming languages, we will implement the coverage-measurement, fault-localization and test-suite minimization tools. We will explain how to develop coverage-based tools with OCCF by writing sample code with OCCF using Visual Studio, as shown in Figure 5.

Additionally, we will introduce a non-free tool, which utilizes OCCF to support Java in our joint research with a company. Because the original tool measured test coverage supporting only C and C++, we extended the tool using OCCF to support Java. To use OCCF from the tool, we added new classes which were designed with Template Method pattern. Our classes insert the instrumentation code providing the structure information of analyzed source code to the tool.



```
        Console.WriteLine(
                "risk, executedAndPassedCount / passedCount, executedAndFailedCount / failedCo
        foreach (var stmt in covInfo.StatementIndexAndTargets) {
            var passedTestCases = testInfo.TestCases.Where(t => t.Passed);
            var executedAndPassedTestCases =
                    passedTestCases.Where(t => t.Statements.Contains(stmt.Item1));
            var failedTestCases = testInfo.TestCases.Where(t => !t.Passed);
            var executedAndFailedTestCases =
                    failedTestCases.Where(t => t.Statements.Contains(stmt.Item1));

            var executedAndPassedCount = executedAndPassedTestCases.Count();
            var passedCount = passedTestCases.Count();
            var executedAndFailedCount = executedAndFailedTestCases.Count();
            var failedCount = failedTestCases.Count();
            var metrics = calcMetricFunc(
                    executedAndPassedCount,
                    passedCount,
                    executedAndFailedCount,
                    failedCount);
            var metricsString = "";
            var delimiter = "";
            foreach (double metric in metrics) {
```

Fig. 5. Screenshot when writing sample code using Visual Studio to develop coverage-based tools