2010 10th International Conference on Quality Software

Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages

Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa
Dept. Computer Science and Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo 1698555, Japan
kazuu@ruri.waseda.jp, washizaki@waseda.jp, fukazawa@waseda.jp

***Abstract*—Test coverage is an important indicator of whether software has been tested sufficiently. However, existing measurement tools for test coverage are associated with several problems such as their cost of development and maintenance, inconsistency and inflexibility in measurement. We propose a framework for consistent and flexible measurement of test coverage, called the Open Code Coverage Framework (OCCF), that supports multiple programming languages. OCCF extracts commonalities from multiple programming languages focusing on only small syntax differences in programming languages using an abstract syntax tree. OCCF provides guidelines to support several test coverage criteria. Moreover, OCCF let users expand features to add user-defined test coverage and new programming language.**

As a result, we reduced the lines of code required to implement measurement tools for test coverage by about 90% and the time to implement a special coverage criterion by 80% or more in an experiment that compared OCCF with conventional tools developed individually without using the framework.

***Keywords*-Software testing; Test coverage; Code coverage; Metrics; Framework;**

I. INTRODUCTION

Test coverage or code coverage (just coverage in the following) is an important measure used in software testing. It refers to the degree to which the source code of a program has been tested and is an indicator of whether software has been tested sufficiently. There are multiple criteria in coverage, such as statement and decision coverage. For instance, statement coverage is the number of statements that have been executed at least once from all the statements. Developers select a suitable criterion according to what purpose their software testing is to be used for [1].

Test coverage measurement tools (just tools in the following) are necessary to measure the coverage of various programs accurately, and tools have become widely available. Many tools are offered for major programming languages (just languages in the following) such as C or Java. However, tools for legacy or minor languages such as COBOL or Squirrel are not readily available are considerably expensive. Moreover, it is more difficult to measure the coverage of newly defined languages such as Go and of existing languages with some changes to language specifications because each existing tool is specific to a certain language specification. Such a situation has driven the need to develop some framework or tool that will correspond to a variety of languages including new languages in the future.

Another driver has been various developments that have been undertaken with multiple languages. For instance, in the development of typical client-server based enterprise systems, client and server applications are developed separately in different languages. This causes few problems in unit testing, which tests each module in isolation, but a number of problems arise in integration testing, which tests the integration of a set of modules. Therefore, tools are required that can consistently support multiple languages.

We propose a framework for consistent and flexible measurement of coverage, called the Open Code Coverage Framework (OCCF), that supports multiple languages [1]. OCCF extracts commonalities from among multiple languages, disregards variability, and lets users focus on only small differences in syntax between languages using an abstract syntax tree (AST) to help the development of tools that can measure the coverage of new languages. Moreover, OCCF provides guidelines to support several coverage criteria and lets users add new coverage criteria and languages.

Using OCCF as a novel framework for developing tools, we reduced the development and maintenance costs of tools and developed sample tools that could consistently and flexibly measure the various coverage criteria of several languages. We implemented a sample tool for C, Java and Python by using OCCF. This tool can measure four coverage criteria. We especially reduced the lines of code (LOCs) required for implementing tools by about 90% and the time to implement a new coverage criterion by 80% or more in an experiment comparing OCCF with conventional individual tools that were non-framework based.

OCCF is now freely available via the Internet [2].

[1]This paper is an extended version of our preliminary short workshop papers presented in [3] and [4], which mainly discuss part of the concept of OCCF. In this paper, there are significant additions including details on OCCF implementation and an important evaluation.

1550-6002/10 $26.00 © 2010 IEEE
DOI 10.1109/QSIC.2010.42

IEEE computer society

II. Problems with conventional approaches

The existing tools can roughly be classified into two types: extension tools that extend programming-language processors and insertion tools that insert measurement code into intermediate language code. Examples of the extension tools include `Statement coverage for Python (SCP)` [5] supporting Python and `gcov` supporting the languages that the GNU Compiler Collection (GCC) [6] supports. `SCP` uses a trace module in the Python standard library. The `gcov` is a subset of GCC that includes measurement code in object files, so `gcov` belongs to insertion tools. Examples of the insertion tools include `Cobertura` [7] and `EMMA` [8] supporting Java, and `NCover` [9] supporting .NET languages. The following summarizes the problems with existing tools.

A. P_1: Cost of new development

Tools are often unavailable for many new, legacy or minor languages due to a lack of community or non-commercial efforts. However, tools for these languages are necessary.

It is generally difficult to implement tools because they have many functions such as syntax and semantics analyzers. It is especially difficult to implement the flexible tools that are mentioned after this that can change the measurement range and elements. The extension tools have to have the processor modified to be sufficiently flexible because they are strongly dependent on the processor. Insertion tools have to connect elements in the source code and in intermediate language programs because they analyze the measurement elements in both source code and intermediate language code. Therefore, a mechanism that can help these new tools to be developed is required.

B. P_2: Cost of maintenance

Language specifications change according to paradigm changes and expanded features. Large changes cause the varieties of syntax to increase and cause semantics to change. For instance, when Java was upgraded to 5.0 from 1.4, new syntax and semantics, such as a foreach statement and the generic type, were added to the language specifications. Also, when Python was upgraded to 3 from 2, a `print` statement changed to just a function call. The existing tools have to be modified for both the processor or the intermediate language and the parser to respond to such changes. Therefore, a mechanism that can help to maintain new tools is required.

C. P_3: Inconsistent measurement

Developers measure the coverage of multiple languages in software development using multiple languages such as the software that is designed based on the client-server model. However, when different tools are used together in integration testing, the measurement results are inconsistent due to the effect of the differences in measurement criteria. According to our investigations, free tools that support multiple languages do not exist, except for `gcov` and `NCover`. We will discuss these points in Section V. Therefore, consistent tools that support multiple languages are required.

D. P_4: Inflexibility in measurement

Coverage results of 100% can indicate software has been sufficiently tested. However, coverage results of less than 100% can also indicate software has been sufficiently tested since this is sufficient if the part deemed to be necessary by the developers has been tested. In addition, the time to run software testing has increased because software-testing techniques such as test-driven development [10] have become quite advanced and the number of test cases has increased. From the perspective of time efficiency, it is better to limit the measurement range and elements.

For instance, Sakata et al. [11] proposed an idea for only measuring functions that are needed in measurements of coverage for components. Therefore, flexible measurements that can limit the measurement range and elements are required to achieve 100% as a sufficient result.

Tools that can freely change the measurement range and elements and that can measure user-defined coverage criteria do not exist, according to our investigation. In addition, many existing tools can only limit the measurement range and only change the size of measurement elements such as statements and blocks. Therefore, flexible tools that support multiple languages are required.

E. P_5: Incomplete measurement

Coverage is measured by using the information on executed elements obtained when software testing is carried out. However, when coverage is measured for an executable binary file, measurement elements that exist in the source code are often ignored because of the difference in semantics between the source code and executable binary file. The optimization function of the compiler often removes dead code such as a private method that is not called or an if statement in which a conditional expression is always evaluated as false.

There is an example of source code including dead code in List 1. `Cobertura` has 100% statement coverage for this source code, but the correct measurement result is 50% statement coverage. Therefore, tools that completely measure coverage are required.

List 1. Sample code of deadCode in Java

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

III. FRAMEWORK FOR MEASURING COVERAGE SUPPORTING MULTIPLE PROGRAMMING LANGUAGES

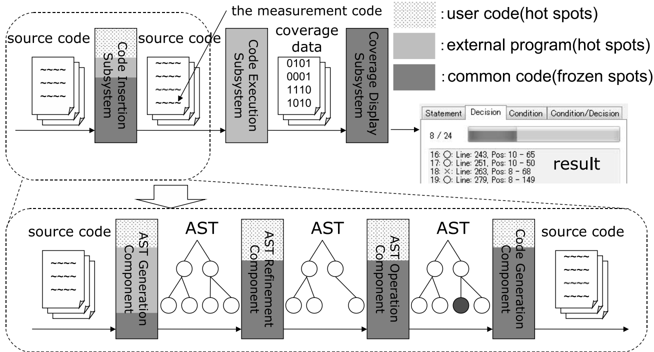

Figure 1. Overview of OCCF

We propose OCCF to support multiple languages and will alleviate or solve the problems previously described.

The framework has reusable software architecture and provides generic design as do some similar applications. The application can be implemented by adding application-specific code as user code to the framework [12].

There is an overview of OCCF and the processing flow in Figure 1. OCCF consists of three subsystems: the code-insertion, code-execution and coverage-display subsystems. Moreover, the code-insertion subsystem consists of four components: the AST-generation, AST-refinement, AST-operation and code-generation components.

The process for measuring coverage involves five steps.

1) Generation of AST from source code
2) Insertion of code for measurement on AST
3) Generation of source code from AST
4) Execution of generated source code and collection of measurement information
5) Display of measurement results from coverage

OCCF inserts the measurement code into the source code, and coverage is measured by executing the program. When OCCF inserts the measurement code, it collects information such as location information on the measurement elements in the source code. The measurement code does not have any side effects except for the processing to collect the coverage information. Thus, OCCF inserts the measurement code without changing the semantics of the original source code.

The source code before insertion is in List 2. The source code after insertion is in List 3.

List 2. Before insertion

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

List 3. After insertion

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

The `stmt_coverage` and `decision_coverage` subroutines measure statement coverage and decision coverage in the example. The `decision_coverage` subroutine returns the evaluation value of the original conditional expression. Therefore, the inserted code has no side effects, and does not change the semantics of the source code.

OCCF provides common code for language-independent processing and provides a design to help user code to be written for language-dependent processing. Moreover, the insertion in AST is simpler than that with other mechanisms such as the extension of programming-language processors, insertion into intermediate language code, or direct insertion into source code because OCCF focuses on the syntax tree structure. In this way, OCCF reduces the cost of development and maintenance. However, OCCF only targets procedural programming languages and impure functional programming languages due to the insertion mechanism.

OCCF supports measurement for new languages by adding user code for language-dependent processing. Moreover, OCCF provides guidelines for implementing coverage criteria by providing classes that insert measurement code. Thus, the part that users should implement is clear. When they appropriately implement this part, they can measure four coverage criteria: statement coverage, decision coverage, condition coverage, and condition/decision coverage.

As OCCF inserts the measurement code before dead code is removed by compiler optimization, it recognizes all measurement elements in the source code. OCCF can completely measure the coverage.

IV. IMPLEMENTATION OF OCCF AND USAGE OF OCCF

We implemented OCCF in .NET Framework 3.5 SP1. OCCF enabled language-specific processing to be implemented by adding user code such as assembly files that ran in .NET Framework 3.5 SP1 or older, or script files in languages supported by Dynamic Language Runtime (DLR) [13]. DLR is a .NET library that provides language services for several different dynamic languages. Moreover, OCCF uses the Managed Extensibility Framework (MEF) [14]. MEF is a .NET library that automatically creates the instance of the class that implements a specific interface and it is annotated with an attribute provided by MEF. Consequently, OCCF eliminates the need for user code that explicitly loads the assembly files and script files and helps to add user code.

We show the implementation of OCCF dividing hot spots and cold spots. We also show the the implementation of a sample tool that measures coverage in Java, C and Python by using OCCF to explain cold spots.

A. Code-insertion subsystem

The code-insertion subsystem consists of the following components: the AST-generation, AST-refinement, AST-operation and code-generation components.

1) AST-generation component: Converts the obtained source code into an AST as an XML document. This component has to parse source code and the parser can be implemented by using existing software, such as compiler compilers and parser libraries.

Cold spots: OCCF provide an `AstGenerator` class that is designed by applying the Template Method pattern [15].

The Template Method pattern reorganizes the processing steps between the coarse-grained process flow and fine-grained concrete processing steps. The former is placed in a superclass method and the latter is placed in subclass methods. The latter is triggered by the former by calling superclass abstract methods that are actually implemented in subclasses.

The `AstGenerator` class calls a parser with a specified command and inputs the result through standard input/output and outputs the AST as an XML document to help users to use a parser of an external program.

Hot spots: Users can implement this component by using the existing software and the inheritance of the `AstGenerator` class giving the command to call the parser.

The sample tool uses SableCC [16] for a Java parser, ANTLR [17] for a C parser and the parser module in the Python standard library for a Python parser. There is a sample user code of this component for Java in List 4.

List 4. JavaAstGenerator.cs

```
1 [Export(typeof(IAstGenerator))]
2 public class JavaAstGenerator : AstGenerator {
3   private static readonly string[] _arguments =
4     new[] { "-jar", "../Java/Java.jar" };
5   protected override string ProcessorPath
6     { get { return "java"; } }
7   protected override string[] Arguments
8     { get { return _arguments; } }
9 }
```

2) AST-refinement component: Changes structure of AST to operate it easily. For instance, this component converts single-line if statements into multi-line if statements. Users have to implement this component for languages that have such grammar by using the AST-operation component.

Cold spots: OCCF provides the `BlockInserter` class that creates a new block and users only pass block symbols.

Hot spots: Users have to implement this component for languages such as C and Java because the measurement code is not easily inserted into some statements such as a single-line if statement. However, this component is not needed for Python because the statement can be inserted before any statement. Users can easily implement this component for C and Java because all if statements can be added to a new block without changing the semantics.

3) AST-operation component: It has roughly three functions: the enumerator, the generator, and inserter. The enumerator finds the corresponding node on the AST of each language to locate the position in which the measurement code has been inserted. The generator is used to generate subtrees corresponding to the measurement code. The inserters are used to insert subtrees of the measurement code into the source code on AST.

Cold spots: There is a class diagram of UML [18] related to the selector in Figure 2. OCCF provides the `ISelector` interface to show the function necessary for the selector. OCCF provides some classes to help users to implement the `ISelector` interface. The `AtomicConditionalTermSelector` class is designed by applying the Template Method pattern. OCCF let users extend it to implement the enumerator that locates the position of all atomic logical terms in conditional expressions. The `SelectorUnion` class integrates some enumeration results. The `SelectorPipe` class enumerates subtrees in other enumeration results. These two classes are designed as Macro Commands by applying the Command pattern [15].

The Command pattern is a design pattern that encapsulates a request and the parameters in an object. A command object that is combined with certain other command objects is called a Macro Command.

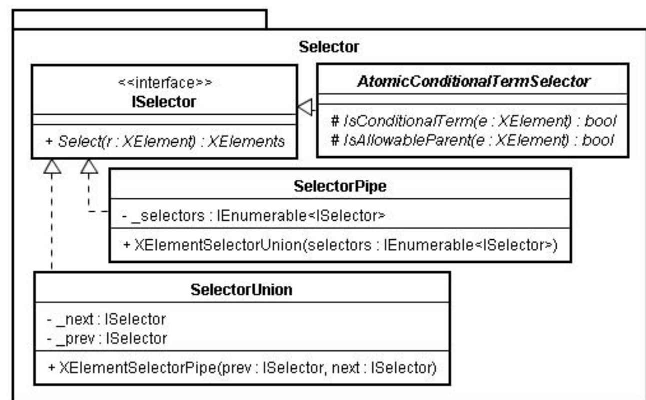

Figure 2. Class diagram of selector in OCCF

OCCF provides an `IStatementGenerator` interface and an `IConditionGenerator` interface to show the function necessary for the selector.

OCCF completely provides the inserter as common code such as a `StatementCoverageInserter` class and a `DecisionCoverageInserter` class. Users pass the instance that implements the `ISelector` to the `Insert`

method of these classes to insert measurement code. OCCF provides these classes as the guidelines for implementing coverage criteria.

Hot spots: Users have to implement the `ISelector` interface for the selector to select statements, conditional expressions in if statements and atomic logical term in conditional expressions in if statements. Users can implement the enumerator by using provided reusable classes.

For example, the enumerator for condition coverage enumerates the atomic logical term elements in if conditional expressions such as the `and_test` and `not_test` nonterminal symbols that has more than three brothers and is not a descendant of the `trailer` in Python grammar. There is a sample user code of the enumerator for the atomic logical term elements of Python in List 5.

List 5. PythonConditionalTermSelector.cs

```
public class PythonConditionalTermSelector
    : AtomicConditionalTermSelector {
  private static string[] TargetNames =
    { "or_test", "and_test" };
  private static string[] ParentNames = { "trailer" };
  protected override bool IsConditionalTerm(XElement e)
    { return TargetNames.Contains(e.Name.LocalName); }
  protected override bool IsAllowableParent(XElement e)
    { return !ParentNames.Contains(e.Name.LocalName); }
}
```

Users have to implement both the callee and caller of the measurement code for the generator. The callee in C/C++ is provided by OCCF so that users can use SWIG to implement it. Users only have to learn to use SWIG or manually port the C/C++ code to the code of the target language. The caller is the code that calls the callee and users simply implement processing that describes the caller code as token element in AST. Users have to implement the `IStatementGenerator` interface for statement coverage and the `IConditionDecorator` interface for decision and condition coverage.

Users can add the new inserters to measure other coverage criteria and flexibly measure coverage.

4) Code-generation component: Converts the obtained AST into source code. When the AST has memorized almost all the tokens for corresponding text in the source code, this component can be simply implemented by adding user code that outputs the tokens as they are without exceptions. This means that the AST-generation component has to add sufficient text information in the source code into the AST to restore the source code with the code-generation component.

Cold spots: OCCF provides a `CodeGenerator` class that is designed by applying the Template Method pattern and scans the AST and outputs the memorized tokens.

Hot spots: Users can easily implement this component by using the `CodeGenerator` class provided by OCCF when the AST has memorized almost all the tokens for corresponding text in the source code. Thus, users should design the AST-generation component to take AST memorization into account. For example, all tokens except for the linefeed and indent are memorized in AST for Python. Consequently, users only have to implement processing that outputs the linefeed and indent to the corresponding terminal nodes for Python. There is a sample user code of this component for Python in List 6.

List 6. PythonCodeGenerator.cs

```
[Export(typeof(ICodeGenerator))]
public class PythonCodeGenerator : CodeGenerator {
  protected override bool TerminalSymbol(XElement e) {
    switch (e.Name.LocalName) {
    case "NEWLINE": WriteLine(); return true;
    case "INDENT":  Depth++; return true;
    case "DEDENT":  Depth--; return true;
    default:        return false;
    }
  }
}
```

B. Code-execution subsystem

The code-execution subsystem executes the program in which the measurement code has been inserted. By executing the program, this subsystem sends the coverage information to the coverage-display subsystem. OCCF supports communication using TCP/IP, shared memory, and file output as the sending mechanism. Although OCCF does not provide this subsystem, users can use any runtime system for the corresponding language. Therefore, they do not need to implement this subsystem.

C. Coverage-display subsystem

The coverage-display subsystem presents the measurement results by analyzing the information received from the code execution subsystem.

The information contains the position and tags. The position expresses the line and column number of the measurement element in the original source code. The tag is a character string that expresses the layered structure. OCCF filters the results of coverage with the package hierarchy, the class hierarchy and other hierarchies with the tags.

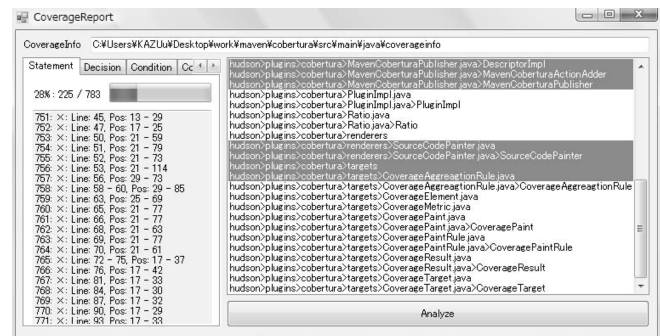

Figure 3. GUI Reporter in coverage-display subsystem

There is a sample window of the coverage-display subsystem in Figure 3. The upper progress bar indicates the

Table I
SUMMARY OF COMPARISON

	OCCF	Cobertura	EMMA	SCP	gcov	NCover
N. coverage criteria	4	2	1	1	3	3
Adding language	yes	no	no	no	yes	no
Adding criteria	yes	no	no	no	no	no
Multiple languages	yes	no	no	no	yes	yes
Flexibility	yes	yes	no	no	no	no
Completeness	yes	no	no	yes	no	yes
Non-commercial	yes	yes	yes	yes	yes	no

coverage ratio. The central text box indicates whether the measurement element was executed in software testing. This sample can output the results as a csv file and an XML file. Therefore, users can customize this subsystem to change the display for all supported languages and can process output files with other tools. OCCF provides this subsystem completely as common code.

V. EVALUATION

We evaluated OCCF by comparing implemented sample that were developed by using OCCF with typical existing tools as described in Section II. The existing tools can roughly be classified into two: those that extend programming-language processors and those that insert measurement code into intermediate language code. Table I summarizes comparison between OCCF and other tools.

A. Reduced cost of new developments

We evaluated the cost of new developments by using the LOCs of the program that inserted the measurement code, by measuring the time to implement a new coverage criterion and by counting the number of supported coverage criteria. **Experiment 1:** We measured the LOCs of the program that inserted the measurement code to evaluate the cost of new developments. The results of comparison we obtained with LOCs are given in 4. There are 1056 LOCs for `Cobertura`, 2031 LOCs for `EMMA`, and 125 for implementing the sample for Java. `Cobertura` uses BCEL [19] to insert measurement code into Java bytecode. BCEL is a library that conveniently provides users with the option to analyze, create, and manipulate Java bytecode. `EMMA` does not use such a library. However, samples were implemented without using a library except for our simple helper methods and the .NET standard library. There are 131 LOCs for `SCP`, and 93 LOCs for the sample with OCCF for Python in Figure 4. The `SCP` uses only used the Python standard library. In addition, there were 221 LOCs for the language independent and reusable parts in the framework. It was difficult to measure the LOCs of the extension tools; however, the cost of development is obviously high. No insertion tools for the source code level exist, as we found. OCCF can support new language at less cost than that in developing new tools by using simple insertion at the source-code level.

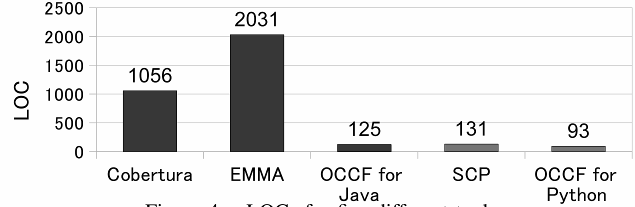

Figure 4. LOCs for five different tools

Table II
NUMBER OF PEOPLE WHO IMPLEMENTED WITHIN FOUR HOURS

	N. people	average time
New coverage for Python 2 with OCCF	4	13.5 min
Change in upgrade to Python 3 with OCCF	4	47.5 min
New coverage for Python 2 with SCP	0	-
Change in upgrade to Python 3 with SCP	0	-

Experiment 2: We experimented on the implementation of a new coverage criterion for Python that is special statement coverage limited to `print` statements. Moreover, we experimented on maintenance that responded to a change in upgrade from Python version 2 to Python version 3. We evaluated the cost of developing a new coverage criterion and the cost of maintenance to change language specifications.

We tested four examinees who were first year Master's degree students studying computer science. We explained our framework for 30 min and provided the AST Generation component for Python 3, which we implemented for them for 25 min. Table II gives the number of people who implemented a tool for Python 2 and responded to a change in upgrade to Python version 3 within four hours. The average time to implement a tool for Python 2 with OCCF is 13.5 min, and the time to respond to a change for Python 3 with OCCF is 47.5 min.

The examinees tried to modify the `SCP` to measure the `print` statement coverage. However, nobody completed it within 240 min. They probably need 10 hr or more. This is because the numerous tools that exist are not highly modularized and it is difficult to find the part of code that has to be modified to extend them. The OCCFs, on the other hand, were highly-modularized and the examinees could easily extend them, e.g., they only implemented the `ISelector` interface in this experiment.

The implementation of samples with OCCF supports statement coverage, decision coverage, condition coverage and condition/decision coverage. However, the number of coverage criteria which other tools support is less than OCCF in Table I. The same functionality could be implemented with fewer LOCs.

Therefore, we succeeded in alleviating the problem (P_1) with high cost of new developments.

B. Reduced maintenance cost

We evaluated the cost of maintenance by assessing the changes to language specifications.

The existing extension tools have to be maintained for the parser and processor. The existing insertion tools also have to be maintained for the parser and intermediate language code. Briefly the existing tools have to be maintained for both the syntax and semantics. However, only the code insertion subsystem has to be maintained in the tool using OCCF. Moreover, only the AST-generation component and the enumerator of the AST-operation component have to be maintained in many cases. As OCCF only focuses on the syntax, the extent of maintenance is limited.

For example, when Java is upgraded to 5.0 from 1.4, OCCF would only be required to respond to locating the foreach statement. `Cobertura` and `EMMA`, on the other hand, are required to respond to generic types as well. Moreover, the experiment on implementing a new coverage for Python and responding to change in upgrade from Python 2 to Python 3 as previously mentioned indicated that it is easier to modify tools using OCCF than with `SCP`.

Therefore, we succeeded in alleviating the problem (P_2) with high cost of maintenance.

C. Consistency in measurement

We evaluated the consistency of measurement by assessing developments with multiple languages.

We measured the coverage for software that was developed in Java and Python as an example. When the software was tested in integration testing, coverage was measured by `Cobertura` and `SCP`. `Cobertura` could measure the statement coverage and decision coverage, but `SCP` could only measure the statement coverage. In this case, coverage with a different criterion was measured, or the same statement coverage was measured. There is a possibility that only coverage that is ineffective can be obtained as an indicator of software testing. However, OCCF could measure coverage with the same criterion, such as decision coverage, for all languages. Effective coverage as an indicator of software testing could be obtained.

The `gcov` and `NCover` can also measure coverage for many languages. However, `gcov` only runs under GCC, e.g., it does not run under Visual C++ [20]. It is difficult to add new languages because `gcov` requires users to imeplement compilers in `GCC`. `NCover` only supports languages that run under the .NET Framework. However, OCCF can measure coverage in any environment where inserted code is running because it does not depend on a specified language processor. Moreover, OCCF lets users add new coverage criteria and languages more easily than `gcov` and `NCover` because it is not just a tool but a framework.

Therefore, we solved the problem (P_3) with inconsistency in measurement.

D. Flexibility in measurement

We evaluated the flexibility of measurement by assessing the change in the measurement range.

The existing tools did not flexibly change the measurement range or elements. `EMMA` and `NCover` could change the measurement range according to only the hierarchy of the package, the class, and the method. `EMMA` could also change the size of measurement elements such as lines, blocks, methods and classes. `Cobertura` could change the measurement elements by using regular expressions. However, OCCF could freely change the measurement range and elements by adding user code that located the position in which the measurement code was inserted. For example, OCCF could insert code to measure statement coverage only for statements that called a specific method. Execution of a program in which measurement code had been inserted thus measured special statement coverage limited only to statements that called a specific method. This coverage could be used in library testing.

Therefore, we solved the problem (P_4) with inflexibility in measurement.

E. Degree of completeness in measurement

We evaluated the degree of completeness in measurement by assessing the measurement for dead code.

`Cobertura` inserts the measurement code into Java bytecode. It does not measure the coverage for dead code because compiler optimization removes dead code in the bytecode. However, OCCF inserts measurement code into the source code before the compiler optimization removes dead code. OCCF can be detected as a part where dead code has not been tested because the information when inserting it remains. For instance, OCCF has correctly 50% statement coverage in List 1 in Section II.

Therefore, we have solved the problem (P_5) with the degree of incompleteness in measurement.

F. Time efficiency

We evaluated the time efficiency by using the time to execute three Java programs presented in a book [21].

Experiment 3: We measured the time to execute three Java programs presented in a book. Measuring coverage decreased the time efficiency for the test because it inserted measurement code into the source code. The execution time when using OCCF was suppressed from 2 to about 10 times the execution time by using TCP/IP communication compared with the former source code, as shown in Figure 5. OCCF is about 10 times faster than `Cobertura`. However, it is 1000 or more times slower than the original source code when used with file output, and it is 10 to about 30 times slower than `Cobertura`. TCP/IP communication is overwhelmingly faster than simple file output.

Therefore, we confirmed that there were no problems with the decrease in execution efficiency of the test when using OCCF with TCP/IP communication.

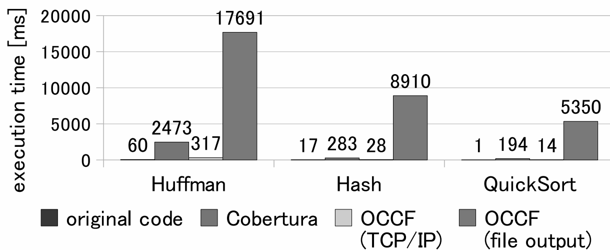

Figure 5. Execution time in software testing

VI. Related work

We discuss the ideas of Kiri et al. [22] and Rajan et al. [23] as other researches. We referred to their study results. Nonetheless, our approach bares resemblance to the following existing techniques.

Kiri et al. proposed the idea of developing a tool that inserts measurement code into source code. Their idea was to measure statement coverage, decision coverage and a special coverage called RC0. RC0 is special statement coverage for only revised statements. However, their idea was to measure only statement coverage and decision coverage because they measured coverage by simply inserting a simple statement. Moreover, even though their idea could be used to measure the coverage of four languages, including Java, C/C++, Visual Basic, and ABAP/4, it did not support any other languages. Conversely, OCCF does not measure RC0. However, OCCF can support new coverage such as RC0 easily by adding user code.

Rajan et al. proposed the idea of specifying the measuring elements with a description style of pointcut that is used in Aspect-oriented programming languages. They demonstrated an example implementation of a tool that supported C#. Measuring elements, such as method calls, if statements, exception handlers and variable writes could be specified. However, the description style that specified the measurement elements was specialized for C#. Therefore, the description style could not be used for other languages that had different paradigms to C#. However, OCCF can measure this coverage with a modified description style that is language independent by easily adding user code.

VII. Conclusion and future work

We proposed OCCF, reduced costs by reusing common code, and obtained consistent measurements by supporting multiple languages, flexible measurements through expanding features, and complete measurements by inserting measurement code into the source code.

We plan to improve OCCF to support non procedure-oriented languages, such as impure functional programming languages. Moreover, we intend to automatically generate AST correspond to the measurement code to reduce development cost further. We believe OCCF can contribute to the development of an implementation of a refactoring tool in language-independent models.

References

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

[2] Kazunori Sakamoto, Open Code Coverage Framework, http://sourceforge.jp/projects/codecoverage/.

[3] Kazunori Sakamoto, et al., "Reporting the Implementation of a Framework for Measuring Test Coverage on Design Pattern", Software Patterns and Quality(SPaQu 2009), 2009.

[4] Kazunori Sakamoto, et al., "A Framework for Measuring Test Coverage Supporting Multiple Programming Languages", First Software Engineering Postgraduates Workshop (SEPoW 2009; In conjunction with APSEC 2009), 2009.

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

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

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

[8] EMMA, http://emma.sourceforge.net/.

[9] NCover, http://www.ncover.com/.

[10] Kent Beck, "Test-Driven Development: By Example", The Addison-Wesley Signature Series, 2003.

[11] 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.

[12] 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.

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

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

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

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

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

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

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

[20] Microsoft, Visual C++, http://msdn.microsoft.com/visualc/.

[21] Haruhiko Okumura, Houki Satoh, Kazuo Turu, Kazuyuki Shudo and Tutimura Nobuyuki, "Algorithm cyclopedia by Java", Gijutuhyoronsya, 2003.

[22] Takashi Kiri, Tatuya Miyoshi, Satoru Kishigami, Tatuo Osato, Tuyoshi Sonehara "About the source code insertion type coverage tool", The 69th Information Processing Society of Japan National Convention, 2003.

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