PAPER

Open Code Coverage Framework: A Framework for Consistent, Flexible and Complete Measurement of Test Coverage Supporting Multiple Programming Languages*

Kazunori SAKAMOTO[†a)], **Fuyuki ISHIKAWA**[††b)], **Hironori WASHIZAKI**[†c)], *and* **Yoshiaki FUKAZAWA**[†d)], ***Members***

SUMMARY Test coverage is an important indicator of whether software has been sufficiently tested. However, there are several problems with the existing measurement tools for test coverage, such as their cost of development and maintenance, inconsistency, and inflexibility in measurement. We propose a consistent and flexible measurement framework for test coverage that we call the Open Code Coverage Framework (OCCF). It supports multiple programming languages by extracting the commonalities from multiple programming languages using an abstract syntax tree to help in the development of the measurement tools for the test coverage of new programming languages. OCCF allows users to add programming language support independently of the test-coverage-criteria and also to add test-coverage-criteria support independently of programming languages in order to take consistent measurements in each programming language. Moreover, OCCF provides two methods for changin the measurement range and elements using XPath and adding user code in order to make more flexible measurements. We implemented a sample tool for C, Java, and Python using OCCF. OCCF can measure four test-coverage-criteria. We also confirmed that OCCF can support C#, Ruby, JavaScript, and Lua. Moreover, we reduced the lines of code (LOCs) required to implement measurement tools for test coverage by approximately 90% and the time to implement a new test-coverage-criterion by over 80% in an experiment that compared OCCF with the conventional non-framework-based tools.

key words: *software testing, test coverage, code coverage, metrics, framework*

1. Introduction

Test coverage or code coverage, which we refer to as just coverage from here on, 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 indicates whether a software has been sufficiently tested or not. There are multiple criteria in a coverage, such as the statement and decision coverage. For instance, statement coverage is the ratio of the statements that have been executed at least once from all the statements. The developers select a suitable criterion according to the purpose of their software testing [1].

Test coverage measurement tools, which will be referred to as just tools from here on, are necessary to accurately measure the kinds of coverage necessary for various programs, and these tools have become widely available. Many tools are provided for major programming languages, which we will refer to as just languages from here on, such as Java. However, tools for legacy or minor languages such as COBOL or Squirrel are not readily available or 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 their language specifications because each existing tool is specific to a certain language specification. These types of situations have driven the need to develop some framework or tool that will correspond to a variety of languages including new languages in the future.

Other drivers have been under development that support multiple languages. For instance, in the development of a typical client-server-based enterprise system, the client and server applications are developed separately in different languages. This causes fewer problems during unit testing, which separately tests each module, but a number of problems have arisen during 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, flexible, and complete coverage measurement called the Open Code Coverage Framework (OCCF), which supports multiple languages*. The framework has a reusable software architecture and has a generic design like some similar applications. The application can be implemented by adding an application-specific code to the framework [13].

OCCF extracts the commonalities from among multiple languages, disregards the variability, and lets users focus on only the small differences in languages using a concrete syntax tree (CST) or abstract syntax tree (AST) to help with the development of the tools that can measure the coverage of the new languages.

Figure 1 outlines the concept behind the simplification, which is provided by OCCF. There are many-to-many re-

Manuscript received August 30, 2010.
Manuscript revised July 23, 2011.

†The authors are with the Dept. Computer Science and Engineering, Waseda University, Tokyo, 169–3018 Japan.
††The author is with National Institute of Informatics, Tokyo, 101–8430 Japan.

*This paper is an extended version of our preliminary short papers presented in [3], [4] and [24], which mainly discussed part of the concept of OCCF. In this paper, there are significant additions including details on the concept for OCCF, and its implementation, limitations, and an important evaluation.

a) E-mail: kazuu@ruri.waseda.jp
b) E-mail: f-ishikawa@nii.ac.jp
c) E-mail: washizaki@waseda.jp
d) E-mail: fukazawa@waseda.jp

DOI: 10.1587/transinf.E94.D.2418

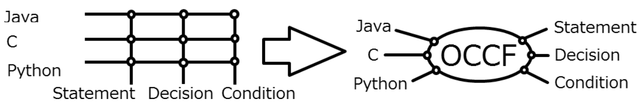


Fig. 1 Simplification of connection between languages and coverage.

lationships between languages and their coverage criteria in the existing tools, and thus all possible combinations must be implemented.

OCCF simplifies the many-to-many relationships into many-to-one relationships between the languages and OCCF, and the one-to-many relationships between OCCF and the coverage criteria. OCCF lets users implement additional languages not depending on coverage criteria and also implement the additional coverage criteria that do not depend on the languages. Such simplification helps users to develop tools and then helps them to freely select their favorite languages and the suitable coverage criteria. OCCF provides a default implementation of the three languages: C, Java, and Python; and four coverage criteria: statement coverage, decision coverage, condition coverage and condition/decision coverage. OCCF lets users add languages that support the default coverage criteria and add coverage criteria that support the default languages. Moreover, OCCF provides two methods to flexibly customize the coverage criteria.

We were able to reduce through experimentation the development and maintenance costs of tools and to develop sample tools that could consistently and flexibly measure the various coverage criteria of several languages by using OCCF as a novel framework for developing tools. In particular, we reduced by approximately 90% the lines of code (LOCs) required for implementing tools and the time to implement a new coverage criterion by 80% or more in an experiment comparing OCCF with the conventional tools that were non-framework based.

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

2. Existing Tools and Conventional Measurement Approaches

There are roughly three approaches for measuring coverage: extending the programming-language processors to add a measurement function, and inserting a measurement code into an executable code or into a source code.

The first approach analyzes both the syntax and semantics of the languages because it parses the source code, analyzes its semantics, and executes it. This approach can adjust the behavior of the program measuring coverage because the programming-language processor decides the program behavior. However, this approach requires a high development cost and it has few measurement features because the programming-language processor is a complex system and it is difficult to add measurement functions. Examples of the tools that use this approach include the `Statement coverage for Python (SCP)` [5], which supports Python, and `gcov`, which supports the languages that the GNU Compiler Collection (GCC) [6] supports. The `SCP` uses a trace module in the Python standard library. `Gcov` is a subset of the `GCC` that includes a measurement code in the object files, and thus, `gcov` also uses the second approach.

The second approach also analyzes both the syntax and semantics of languages because it parses the source code to show the users part of the covered source code and analyzes the executable code in order to insert a measurement code into the executable code. This approach also requires a large development cost since it requires an analysis of both the source and executable codes. Moreover, this approach is unable to adjust well with the program behavior into which the measurement code is inserted because the behavior of the executable code is influenced by the compilers and execution environment. Some examples of the tools that use this approach include `Cobertura` [7], which supports Java, `EMMA` [8] for Java, and `NCover` [9] .NET languages. These tools include measurement codes in the intermediate language codes.

The last approach analyzes only the syntax because it parses the source code in order to decide where to insert the measurement code by analyzing the grammar of the given language. However, this approach cannot adjust the program behavior into which the measurement code is inserted as well as the second approach can. According to our investigations, the tools that use this approach are not widespread although this approach is known.

There is a narrow commonality in both the first and second approaches among the measurement features because these approaches strongly depend on each language. There is also a narrow commonality in the last approach among the measurement features because this approach uses an adhoc processing that focuses on only the grammar. Moreover, it is difficult for the last approach to measure the coverage flexibly without having the semantics of the language.

3. Problems with Conventional Measurement Approaches

3.1 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.

There are many combinations of languages and coverage criteria and it is difficult to implement all the combinations. It is also difficult to extract the commonality in the conventional approaches as already described. Moreover, it is especially difficult to implement the flexible tools that are mentioned after this that can change the measurement range and elements. Therefore, a mechanism that can help to develop these new tools is required.

3.2 P_2: Cost of Maintenance

Language specifications change according to the paradigm changes and expanded features. Large changes cause the varieties of syntax to increase and cause the 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 a generic type, were added to the language specifications. In addition, when Python was upgraded to 3 from 2, a `print` statement changed to just a function call.

When taking into consideration the existing tools, the range of the features needing to be maintained need to be expanded because the existing tools analyze both the syntax and semantics.

Therefore, a mechanism that can help to maintain new tools is required.

3.3 P_3: Inconsistency in Measurement

Developers measure the coverage of multiple languages during the development of software involving multiple languages, such as software designed on the basis of the client-server model. However, when different tools are used together during integration testing, the measurement results are inconsistent because of the effect of differences in the measurement criteria. These differences, such as differences of whether tools do count logical statements or line statements, are not described in the tool specifications (manual documents). Developers may analyze measurement results erroneously and may incorrectly conclude test adequacy owing to the lack of knowledge about the differences.

Suppose, for instance, developers obtained a measurement result showing 100% statement coverage for a program written in Java and Python by using `EMMA` and `SCP`, respectively. `EMMA` and `SCP` support only statement coverage. `EMMA` divides a ternary expression (i.e. `condition ? true-expression: false-expression`) into two statements and can determine whether both branches of the ternary expression have been executed. `SCP`, on the other hand, does not divide ternary expressions and cannot determine whether both branches have been executed. When developers recognize incorrectly that both `EMMA` and `SCP` divide ternary expressions into two statements, they may judge erroneously that all ternary expressions have been well tested. However, untested ternary expressions may exist in the Python program.

Similarly, there are many differences, such as a difference of whether tools do count conditional expressions without control-flow statements as conditional branching, in coverage criteria of existing tools that can mislead developers.

According to our investigations, except for `gcov` and `NCover`, free tools that support multiple languages do not exist. We will discuss these points in Sect. 6. Therefore, consistent tools supporting multiple languages are required.

3.4 P_4: Inflexibility in Measurement

Coverage results that are 100% indicate that a piece of software has been sufficiently tested. However, coverage results that are less than 100% can also indicate software has been sufficiently tested since this is sufficient if the part deemed 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 [11], have become quite advanced and the number of test cases has increased. From the perspective of the time efficiency, it is better to limit the measurement range and the elements, such as those for only a specific method and the elements, such as only assignment statements.

For instance, Sakata et al. [12] proposed an idea for only measuring the functions that are needed in the measurements of the coverage for the components. Therefore, flexible measurements that can limit the measurement range and elements are required to achieve a 100% sufficient result.

Tools that can freely change the measurement range and elements and that can measure the 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 the measurement elements, such as the statements and blocks. Therefore, the flexibility to allow for user-defined coverage criteria is required (with support for multiple languages).

3.5 P_5: Incompleteness in Measurement

Coverage is measured by using the information on the executed elements obtained when the software testing is carried out. However, when the coverage is measured for an executable binary file, the existing measurement elements in the source code are often ignored because of the difference in semantics between the source code and executable binary file. The optimization facility of the compiler often removes the 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 a source code that includes dead code in Fig. 2. `Cobertura` has a 100% statement coverage for this source code, but the correct measurement result is only a 50% statement coverage. A dead code is undesirable because the cost of the maintenance increases when the developers cannot judge whether the description of a dead code is intentional or not. The measurement results of the coverage should express the existence of the dead code. Therefore, tools that completely measure the coverage are required.

```java
public class DeadCode {
  public static void main(String[] args) {
    System.out.println("main");
    if (false) {
      System.out.println("deadcode");
    }
  }
}
```

Fig. 2 Sample code of Dead code in Java.

4. Framework for Measuring Coverage Supporting Multiple Programming Languages

We propose OCCF to support multiple languages, and which will alleviate the problems outlined in Sect. 3.

4.1 Measurement Approach of OCCF

OCCF inserts a measurement code into the source code using AST, and the coverage is measured by executing the program. Our approach analyzes the syntax and the required part of the semantics because our approach parses the source code to get the AST and locates the position of the node where the measurement code is inserted. There is wide commonality in our approach among the measurement features because the insertion processing using the AST in each language is similar.

The source code before it is inserted is outlined in Fig. 5. The source code after being inserted is outlined in Fig. 6. The `stmt_record` and `decision_record` subroutines measure the statement coverage and decision coverage in the example. The `decision_record` subroutine returns the evaluation value of the original conditional expression. OCCF inserts the `stmt_record` into each statement and each variable initializer to measure statement coverage. OCCF also inserts the `decision_record` into each conditional expression of the control-flow statement and inserts the `stmt_record` into each case clause of the switch statement to measure the decision coverage, condition coverage, and condition/decision coverage.

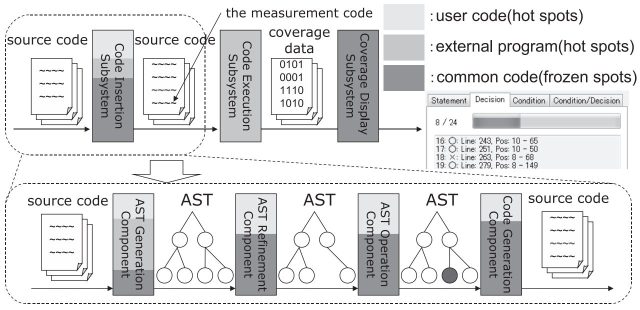


Fig. 3 Overview of OCCF.

The measurement code does not have any side effects except for the processing to collect the coverage information and the changing time behavior. This means that there is a possibility that the semantics of a program using a thread might change. However, it seems that this change can be disregarded by the change due to the execution environment. Therefore, the measurement code has no side effects, and does not change the semantics of the source code.

4.2 Overview of OCCF

An overview of OCCF and the processing flow in shown in Fig. 3. OCCF adopts the general architecture of the measurement tool that used the insertion approach of the measurement code. OCCF consists of three subsystems: the code-insertion, code-execution, and coverage-display subsystems. Moreover, OCCF reduces the size of the code-insertion subsystem for reuse. The code-insertion subsystem consists of three components: the AST-generation, AST-operation, and the code-generation components.

The process for measuring coverage includes six steps for expanding the code-insertion subsystem.

1. Generation of AST from source code
2. Refinement of AST

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

Fig. 5 Before insertion.

```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");
  }
}
```

Fig. 6 After insertion.

	User code	Common code	External program
Code Insertion			
AST Generation	indicates location and arguments of parser	calls parser	parser library or compiler compiler
AST Refinement	indicates position where block is inserted	inserts block	none
AST Operation	indicates position where measurement code is inserted	help indicating	none
	generates subtree corresponding to measurement code	inserts measurement code	none
	Implements callee of measurement code in each language	none	SWIG[23]
Code Generation	outputs special tokens not memorized in AST	outputs tokens memorized	none
Code Execution	none	none	programming-language processor
Coverage Display	none	shows coverage result	none

Fig. 4 Overview of relation between user code, common code and external program.

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

OCCF provides a common code for the language-independent processing that operates the AST in the similar structures and also provides a design to help user codes to be written for language-dependent processing. There is a relation between the common code, user code, and external program listed in Fig. 4. In this way, OCCF reduces the cost of development and maintenance in order to solve P_1 and P_2. However, OCCF only targets the procedural programming languages and impure functional programming languages due to its insertion approach.

OCCF supports the measurement of new languages and the coverage criteria by adding in a user code. Users can implement the AST-generation and code-generation components, and the part of the AST-operation component for adding new languages. When they appropriately implement them, they can measure four default coverage criteria: statement coverage, decision coverage, condition coverage, and condition/decision coverage for the new languages. Users can also implement the part of the AST-operation component for adding new coverage criteria. When they appropriately implement them, they can measure the new coverage criteria for default languages such as C, Java, and Python. We also confirmed that the source code and AST are mutually converted in several languages: Ruby, JavaScript, and Lua by using OCCF in the same way that it implements default languages. In this way, OCCF consistently supports multiple languages in order to solve P_1 and P_3.

OCCF provides two methods for limiting the measurement range and elements: the filter condition described by XPath and the adding of new coverage criteria. OCCF supports the filter condition in which the parent/child/sibling nodes include/exclude the elements that are described by the XPath. OCCF also supports the addition of new coverage criteria to freely change the measurement range and elements. In this way, OCCF flexibly measures the coverage in order to solve P_4.

Since OCCF inserts the measurement code before the dead code is removed by the compiler optimization, it recognizes all the measurement elements in the source code. In this way, OCCF completely measures the coverage in order to solve P_5.

5. Implementation of OCCF

We implemented OCCF in .NET Framework 4.0. OCCF enabled language-specific processing to be implemented by adding a user code, such as the assembly files that ran in .NET Framework 4.0 or older or the script files in the languages supported by the Dynamic Language Runtime (DLR) [14]. The DLR is a .NET library that provides language services for several different dynamic languages. Moreover, OCCF uses the Managed Extensibility Framework (MEF) [15]. The MEF is a .NET library that automatically creates an instance of the class that implements a specific interface and it is annotated with an attribute provided by the MEF. Consequently, OCCF eliminates the need for a user code that explicitly loads the assembly files and script files and helps to add in the user code.

We will show the implementation of OCCF by dividing the hot spots from the cold spots and also show the implementation of a sample tool.

5.1 Code-Insertion Subsystem

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

5.1.1 AST-Generation Component

converts the obtained source code into an AST as an XML document. This component has to parse the source code, and the parser can be implemented by using the existing software, such as the compilers and parser libraries. This component may generate what kind of syntax tree if the following components operate correctly, and OCCF does not limit the schema of the syntax tree.

Cold spots: OCCF provides an `AstGenerator` class that is designed using the Template Method pattern [16].

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 the superclass method and the latter is placed in the subclass methods. The latter is triggered by the former by calling on the superclass abstract methods that are actually implemented in the subclasses.

A class diagram of a UML [17] that is related to this component is shown in Fig. 7. The `AstGenerator` and `AntlrAstGenerator` are an abstract class that is provided by OCCF and designed using the Template Method pattern.

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

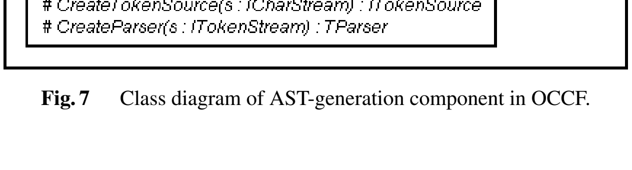

AstGenerator
<<interface>>
IAstGenerator
+ Generate(path : string) : XElement
ExternalAstGenerator
# ProcessorPath() : string
# Arguments() : string[]
TParser
AntlrAstGenerator
# Refine() : void
# CreateTokenSource(s : ICharStream) : ITokenSource
# CreateParser(s : ITokenStream) : TParser

Fig. 7 Class diagram of AST-generation component in OCCF.

```
[Export(typeof(IAstGenerator))]
public class PythonAstGenerator : AstGenerator {
  private static readonly string[] _arguments =
  new[] { "ParserScripts/st2xml.py" };
  protected override string ProcessorPath
  { get { return "C:/Python31/python.exe"; } }
  protected override string[] Arguments
  { get { return _arguments; } }
}
```

Fig. 8 PythonAstGenerator.cs.

AntlrAstGenerator calls the parser, which is generated by ANTLR [18], and outputs the AST as an XML document to help the users to use the ANTLR. Therefore, the users only have to implement the parser and the caller of the parser by using the existing software.
Hot spots: Users can implement this component by using the existing software and the inheritance of the AstGenerator class by giving the command to call the parser.

The sample tool uses the ANTLR for a Java and C parser, and the parser module in the Python standard library for a Python parser. A sample of the user code of this component for Python is outlined in Fig. 8.

5.1.2 AST-Refinement Component

changes structure of AST in order to more easily operate it. 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 structures by using the AST-operation component.
Cold spots: OCCF provides the BlockInserter class that creates a new block. The users only have to pass the block symbols to it.
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 of the statements, such as single-line 'if' statements. However, this component is not required 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. The sample user code of this component for C is provided in Fig. 9.

5.1.3 AST-Operation Component

This component has roughly four functions: the selector, generator, inserter, and tagger. The selector finds the corresponding node on the AST for each language to locate the position in which the measurement code has been inserted. The generator generates the subtrees corresponding to the measurement code. The inserters insert the subtrees of the measurement code into the source code on the AST. The tagger provides the place information of the measurement element in the source code as a tag.
Cold spots: A class diagram of the selector is shown in

```
[Export(typeof(ISelector))]
public class CLackingBlockSelector : ISelector {
  private static readonly string[] ParentNames =
  { "selection_statement", "iteration_statement" };
  private static readonly string[] StatementNames =
  { "statement" };
  protected override IEnumerable<XElement>
      SelectContainingNull(XElement root) {
    return root.Descendants().Where(e =>
        ParentNames.Contains(e.Name.LocalName))
      .Select(e => e.Elements()
        .FirstOrDefault(e2 =>
          StatementNames.Contains(e2.Name.LocalName)));
  }
}
```

Fig. 9 CLackingBlockSelector.cs.

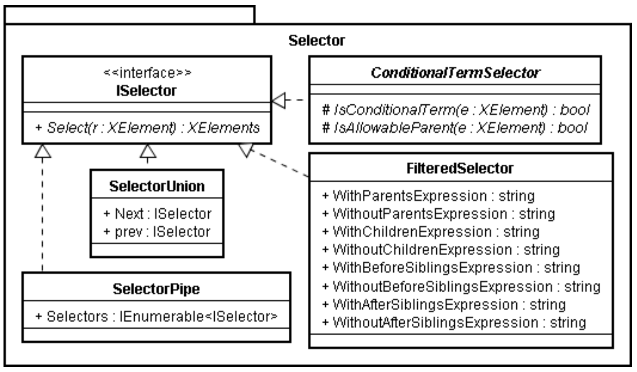

Selector
<<interface>>
ISelector
+ Select(r : XElement) : XElements
ConditionalTermSelector
# IsConditionalTerm(e : XElement) : bool
# IsAllowableParent(e : XElement) : bool
SelectorUnion
+ Next : ISelector
+ prev : ISelector
FilteredSelector
+ WithParentsExpression : string
+ WithoutParentsExpression : string
+ WithChildrenExpression : string
+ WithoutChildrenExpression : string
+ WithBeforeSiblingsExpression : string
+ WithoutBeforeSiblingsExpression : string
+ WithAfterSiblingsExpression : string
+ WithoutAfterSiblingsExpression : string
SelectorPipe
+ Selectors : IEnumerable<ISelector>

Fig. 10 Class diagram of selector in OCCF.

```
public abstract class ConditionalTermSelector:ISelector {
  protected abstract bool IsConditionalTerm(XElement e);
  protected abstract bool IsAllowableParent(XElement e);
  public IEnumerable<XElement> Select(XElement root) {
    var targetParents = root.Descendants()
      .Where(IsConditionalTerm)
      .Where(e => e.Elements().Count() >= 3)
      .Where(e => e.ParentsWhile(root)
        .All(IsAllowableParent));
    var targets = targetParents
      .SelectMany(e => e.Elements().OddIndexElements());
    // a == b&&(a == c||a == d) => a == b,a == c,a == d
    var atomicTargets = targets.Independents().ToList();
    atomicTargets.Sort((e1,e2)=>e1.IsBefore(e2) ? -1:1);
    return atomicTargets;
  }
}
```

Fig. 11 ConditionalTermSelector.cs.

Fig. 10. 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 ConditionalTermSelector class is outlined in Fig. 11 is designed by using the Template Method pattern. OCCF lets users extend it in order to implement the selector for locating the position of all the atomic logical terms in the conditional expressions. The SelectorUnion class integrates some of the selection results. The SelectorPipe class selects the subtrees from other selection results. These two classes are designed as Macro Commands by using the Command pattern [16].

The Command pattern is a design pattern that encapsu-

```
[Export(typeof(ISelector))]
public class PythonConditionalTermSelector
    : ConditionalTermSelector {
  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); }
}
```

Fig. 12 PythonConditionalTermSelector.cs.

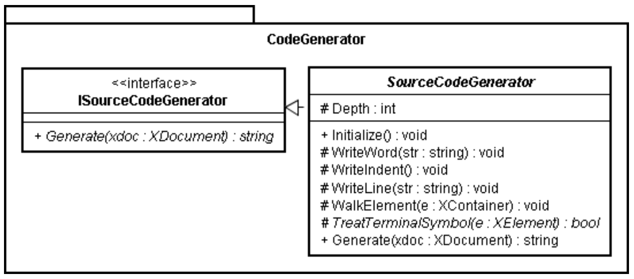

CodeGenerator
<<interface>>
ISourceCodeGenerator
+ Generate(xdoc : XDocument) : string
SourceCodeGenerator
# Depth : int
+ Initialize() : void
# WriteWord(str : string) : void
# WriteIndent() : void
# WriteLine(str : string) : void
# WalkElement(e : XContainer) : void
# TreatTerminalSymbol(e : XElement) : bool
+ Generate(xdoc : XDocument) : string

Fig. 13 Class diagram of code-generator component in OCCF.

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

In addition, OCCF provides a `FilteredSelector` class, which limits the measurement range and element by using the filter condition described by XPath.

OCCF provides an `INodeGenerator` interface to show the function necessary for the generator.

OCCF completely provides the `CoverageInserter` class as a common code for the inserter. Users pass the instance that implements the `ISelector` and the `INodeGenerator` to the Insert method of these classes to `Insert` the measurement code. OCCF provides this class to support the default coverage criteria as guidelines for implementing the coverage criteria.

OCCF provides an `ITagger` interface to show the function necessary for the tagger.

Hot spots: Users have to implement the `ISelector` interface for the selector to select the statements, the conditional expressions in control-flow statements such as ‘if’, ‘for’, ‘while’ statements and ternary expressions, and the atomic logical term in the conditional expressions in the control-flow statements. Users can implement the selector by using the reusable classes that are provided.

For example, the selector for the condition coverage selects the atomic logical term elements in the conditional expressions of the control-flow statements, such as the `and_test` and `not_test` nonterminal symbols, that have more than three brothers and is not a descendant of the `trailer` in the Python grammar. There is a sample user code of the selector for the atomic logical term elements of Python is outlined in Fig. 12.

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 the 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 the users simply implement the processing that describes the caller code as a token element in the AST. Users have to implement the `INodeGenerator` interface as the caller of the measurement code.

Users can implement all the AST-generation, Code-generation, and AST-operation component except for the inserter to add new languages. Users can also implement the inserter that uses the existing selectors for languages to add new coverage criteria. At present, OCCF provides only the necessary selector for the default coverage criteria. For example, when users modify an inserter to measure the modified condition/decision coverage (MCDC), they have to implement the selectors that locate the terms of the logical disjunction and logical production separately for each language. OCCF simplifies the many-to-many relationships between the languages and the coverage criteria because the inserter does not depend on the other components.

Users can implement the tagger to narrow down the measurement results by using the tag. For example, the user code gets the class and method names of the measurement elements by scanning the parent nodes of the measurement elements in the AST. Users can narrow down the measurement results for the GUI when implementing the tagger although the users may not implement the tagger.

5.1.4 Code-Generation Component

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

Cold spots: A class diagram of the code-generator component is shown in Fig. 13. OCCF provides a `CodeGenerator` class that is designed by using 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 the corresponding text in the source code. Thus, users should design the AST-generation component to take the AST memorization into account. For example, all the tokens except for the linefeed and indent are memorized in the AST for Python. Consequently, users only have to implement the processing that outputs the linefeed and indent to the corresponding terminal nodes for Python. A sample user code of this component for Python is outlined in Fig. 14.

5.2 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 communications using TCP/IP, the shared memory, and the file output as the sending mechanisms. 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.

5.3 Coverage-Display Subsystem

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

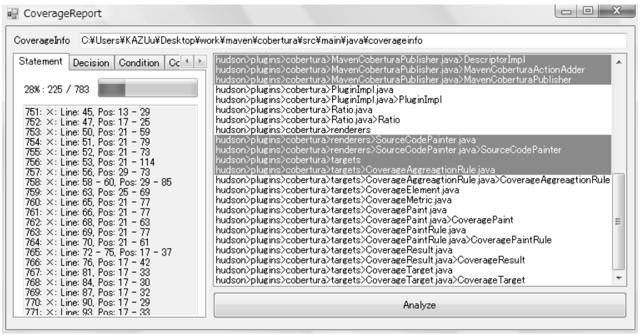
The information contains the position and tags. The position expresses the line and column number of the measurement element in the original source code. The tag is a character string that expresses the layered structure. OCCF filters the results of the coverage with the package hierarchy, the class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

There is a sample window of the coverage-display subsystem shown in Fig. 15. The upper progress bar indicates the coverage ratio. The central text box indicates whether the measurement element was executed during software testing and also shows the position.

A sample can output the results as a csv and an XML file. Therefore, users can customize this subsystem to change the display for all the supported languages and can process the output files using other tools. OCCF provides this entire subsystem as a common code.

```
[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;
    }
  }
}
```

Fig. 14 PythonCodeGenerator.cs.



```
C:¥Users¥KAZUu¥Desktop¥work¥maven¥cobertura¥src¥main¥java¥coverageinfo

28% : 225 / 783

751: ×: Line: 45, Pos: 13 - 29
752: ×: Line: 47, Pos: 17 - 25
753: ×: Line: 50, Pos: 21 - 59
754: ×: Line: 51, Pos: 21 - 79
755: ×: Line: 52, Pos: 21 - 73
756: ×: Line: 53, Pos: 21 - 114
757: ×: Line: 56, Pos: 29 - 73
758: ×: Line: 58 - 60, Pos: 29 - 85
759: ×: Line: 63, Pos: 25 - 69
760: ×: Line: 65, Pos: 21 - 77
761: ×: Line: 66, Pos: 21 - 77
762: ×: Line: 68, Pos: 21 - 63
763: ×: Line: 69, Pos: 21 - 77
764: ×: Line: 70, Pos: 21 - 61
765: ×: Line: 72 - 75, Pos: 17 - 37
766: ×: Line: 76, Pos: 17 - 42
767: ×: Line: 81, Pos: 17 - 33
768: ×: Line: 84, Pos: 17 - 30
769: ×: Line: 87, Pos: 17 - 32
770: ×: Line: 90, Pos: 17 - 29
771: ×: Line: 93, Pos: 17 - 33

hudson>plugins>cobertura>MavenCoberturaPublisher.java>DescriptorImpl
hudson>plugins>cobertura>MavenCoberturaPublisher.java>MavenCoberturaActionAdder
hudson>plugins>cobertura>MavenCoberturaPublisher.java>MavenCoberturaPublisher
hudson>plugins>cobertura>PluginImpl.java
hudson>plugins>cobertura>PluginImpl.java>PluginImpl
hudson>plugins>cobertura>Ratio.java
hudson>plugins>cobertura>Ratio.java>Ratio
hudson>plugins>cobertura>renderers
hudson>plugins>cobertura>renderers>SourceCodePainter.java
hudson>plugins>cobertura>renderers>SourceCodePainter.java>SourceCodePainter
hudson>plugins>cobertura>targets
hudson>plugins>cobertura>targets>CoverageAggreagtionRule.java
hudson>plugins>cobertura>targets>CoverageAggreagtionRule.java>CoverageAggreagtionRule
hudson>plugins>cobertura>targets>CoverageElement.java
hudson>plugins>cobertura>targets>CoverageMetric.java
hudson>plugins>cobertura>targets>CoveragePaint.java
hudson>plugins>cobertura>targets>CoveragePaint.java>CoveragePaint
hudson>plugins>cobertura>targets>CoveragePaintRule.java
hudson>plugins>cobertura>targets>CoveragePaintRule.java>CoveragePaintRule
hudson>plugins>cobertura>targets>CoverageResult.java
hudson>plugins>cobertura>targets>CoverageResult.java>CoverageResult
hudson>plugins>cobertura>targets>CoverageTarget.java
hudson>plugins>cobertura>targets>CoverageTarget.java>CoverageTarget
```

Fig. 15 GUI Reporter in coverage-display subsystem.

6. Evaluation

We evaluated OCCF by comparing implemented samples that were developed by using OCCF with standard tools that are used as described in Sect. 3. There are two main types of standard tools, those that extend the programming-language processors and those that insert a measurement code into the intermediate language code. Table 1 provides a comparison between OCCF and the other tools.

Experiment 1: We obtained measurement results for statement coverage and condition coverage using OCCF and the state-of-the-art tools to confirm that OCCF measures coverage as accurately as the state-of-the-art tools. We targeted three Java programs presented in a book [20] that use typical programming constructors and algorithms. We translated these Java programs into C and Python.

Table 2 lists the measurement results for each program. The columns with the headings "statement" and "condition" indicate the measurement results for statement coverage and condition coverage, respectively. The measurement results are described as "XX%(YY/ZZ)". XX, YY, and ZZ indicate measurement results as a percentage, and numbers of executed measurement elements and total executable measurement elements, respectively. We adopted `gcov` as the state-of-the-art tool for C, `Cobertura` as the state-of-the-art tool for Java and `SCP` as the state-of-the-art tool for Python because these are well accepted. "-" in Table 2 indicates the tools can not measure condition coverage.

The measurement results cannot be directly compared. There are three differences between OCCF and the state-of-

Table 1 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

Table 2 Comparison of measurement results.

	statement of OCCF	statement of tools	condition of OCCF	condition of tools
BTree (C)	57%(92/162)	55%(80/145)	55%(45/82)	55%(45/82)
LZ (C)	95%(161/169)	97%(114/117)	92%(79/86)	92%(79/86)
BoyerMoore (C)	57%(20/35)	67%(20/30)	47%(14/30)	47%(14/30)
All (C)	81%(29/36)	81%(29/36)	70%(14/20)	70%(14/20)
BTree (Java)	63%(102/162)	62%(78/124)	54%(45/84)	52%(45/86)
LZ (Java)	97%(181/186)	100%(113/113)	92%(79/86)	92%(79/86)
BoyerMoore (Java)	64%(29/45)	66%(24/36)	53%(15/28)	53%(15/28)
All (Java)	86%(38/44)	86%(38/44)	69%(18/26)	69%(18/26)
BTree (Python)	66%(114/173)	65%(99/152)	46%(25/54)	-
LZ (Python)	99%(140/141)	100%(130/130)	93%(39/42)	-
BoyerMoore (Python)	78%(42/54)	78%(35/45)	41%(9/22)	-
All (Python)	81%(29/36)	81%(29/36)	50%(4/8)	-

the-art tools.

First, OCCF measures statement coverage with respect to each logical statement, whereas the state-of-the-art tools measure statement coverage with respect to each line. However, we think that the statement coverage where the number of lines does not influence is more accurate than the statement coverage where the number of lines influences.

Second, OCCF does not count conditional expressions without control-flow statements as conditional branching, whereas `Cobertura` does count all conditional expressions such as "`cond = a > 1`". However, we think that the condition coverage which does not count these conditional expressions is more accurate than the condition coverage which does count these conditional expressions because conditional expressions without control-flow statements are not conditional branching in a narrow sense.

Last, OCCF does not count abbreviated default constructors as a statement, whereas `Cobertura` does count abbreviated default constructors. There are two ideas. We can think that abbreviated default constructors are not statements because abbreviated default constructors do not appear in source code. We can also think that abbreviated default constructors are statements because abbreviated default constructors are executed by programming-language processors. Moreover, OCCF, `gcov` and `Cobertura` can measure condition coverage, whereas `SCP` cannot measure condition coverage.

However, OCCF can obtain the same measurement results by adding a user code that measures statement coverage with respect to each line, a user code that does count all conditional expressions, and a user code that supplements default constructors and inserts the measurement code. We actually obtained the same measurement results.

Measurement tools have to do count the following measurement elements for C to measure statement coverage accurately: expression, goto, continue, break, return, if, switch, while, do-while and for statements. They also have to do count the following measurement elements for Java: expression, continue, break, return, assert, throw, if, switch, while, do-while, for, enhanced for, try and synchronized statements. They also have to do count the following measurement elements for Python: expression, assignment, assert, pass, del, print, yield, with, function-definition and class-definition statements. Moreover, OCCF and the state-of-the-art tools do count variable initializers as statements because variable initializers can contain instruments as expressions.

Measurement tools have to do count the following measurement elements for C to measure condition coverage accurately: conditional terms that are separated logical operators in if, while, do-while, for statements and ternary expressions; and case clauses in switch statements. They also have to do count additionally the following measurement elements for Java and Python: enhanced for statements. Moreover, Python does not have switch, for and do-while statements.

The rows with the headings "All" indicate the measurement results of source code that contains all the above-mentioned measurement elements and that is written by us. OCCF and the state-of-the-art tools obtained the same measurement results.

Therefore, we confirmed that OCCF measures coverage accurately.

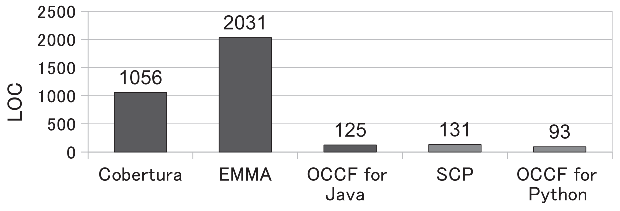


Fig. 16 LOCs for five different tools.

6.1 Reduced Cost of New Developments

We evaluated the cost of new developments by comparing the LOCs of the program that inserted the measurement code, by measuring the time to implement two coverage criteria and by counting the number of supported coverage criteria.

Experiment 2: We obtained the LOCs of the program that inserted the measurement code to evaluate the cost of new developments. The results of the comparison of the LOCs are given in Fig. 16 To implement the sample for Java, 1056 LOCs were required for `Cobertura`, 2031 LOCs were required for `EMMA`, and 125 LOCs were required for OCCF. `Cobertura` uses `BCEL` [19] to insert the measurement code into the Java bytecode. `BCEL` is a library that provides users with the convenient feature to analyze, create, and manipulate the Java bytecode. `EMMA` does not use such a library. However, the samples were implemented without using a library with the exception of our simple helper methods and the .NET standard library. To implement the sample for Python, 131 LOCs were required for `SCP` and 93 LOCs were required for OCCF. `SCP` uses only the Python standard library. In addition, 221 LOCs were required for the language independent and reusable parts in the framework. It was difficult to obtain the LOCs of the extension tools; however, the cost of development was clearly high. In addition, we did not find any insertion tools for the source code level. By using simple insertion in the source code level, OCCF can support new languages at a lower cost than that required to develop new tools.

Experiment 3: We carried out an experiment on the implementation of statement coverage and decision coverage for C because C is a major and practical language. We evaluated the cost of developing two coverage criteria.

We tested five master's degree students studying computer science, who are able to read and write C and Java code. We explained our framework to them in 50 min and then provided them with the AST-generation and the AST-refinement components for C, which we implemented for them in 40 min. Table 3 lists the number of people who

Table 3 Number of people who implemented a coverage tool successfully and average time required to implement.

	N. people	average time
Statement coverage for C with OCCF	5	24.8 min
Decision coverage for C with OCCF	5	53.4 min
Statement coverage for C with GCC	0	-
Decision coverage for C with GCC	0	-
New decision coverage for Java with OCCF	5	34.2 min
New decision coverage for Java with Cobertura	0	-
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 (Nobody)	-
Change in upgrade to Python 3 with SCP	0 (Nobody)	-

implemented the coverage tools for C within 300 min. The average time required to implement the statement coverage tool for C using OCCF was 24.8 min, and the time required to implement the decision coverage tool for C using OCCF was 53.4 min. The examinees attempted to extend GCC to measure the two coverage criteria. However, nobody completed this task within 300 min. "-" in Table 3 indicates these task were not completed and they required more than 300 min.

Existing compiler frameworks such as GCC provide features to add support for new languages. However, there are differences between these compiler frameworks and OCCF regarding cold spots and hot spots. The compiler frameworks provide fewer cold spots to measure coverage than OCCF because they do not specialize in the development of coverage tools and they require developers to consider how to measure coverage. For example, OCCF provides cold spots to insert measurement code into the location selected by the selector. Thus GCC requires more hot spots than OCCF. For example, GCC requires both a semantic analyzer and a syntax analyzer to add a new language. OCCF, on the other hand, requires a syntax analyzer and only the part of a semantic analyzer related to measurement elements. Moreover, OCCF can reuse an existing parser such as the frontends of GCC as AST-generation component. Experiment 3 indicated that a developer can develop a coverage tool using the OCCF more easily than using GCC.

The implementation of samples using OCCF supports the measurement of the statement coverage, decision coverage, condition coverage, and condition/decision coverage. However, the number of coverage criteria that the other tools support was less than that of OCCF according to Table 1; thus, the same functionality was implemented with fewer LOCs.

Therefore, we succeeded in alleviating the problem (P_1) of the high cost of new developments for a given language.

6.2 Reduced Maintenance Cost

We evaluated the maintenance cost by measuring the times required to extend existing decision coverage for Java, to implement special coverage for Python version 2, and to update special coverage from Python version 2 to Python version 3, and by assessing the changes to the language specifications.

Experiment 4: We carried out an experiment on the implementation of a new coverage criterion for Java that was a special decision coverage that takes try statements for exception handlers in Java as conditional branching. This special decision coverage is required to judge a catch block that has no statement was executed. However, the existing tools cannot measure decision coverage in consideration of try statements. We evaluated the cost of maintenance required to extend the existing decision coverage.

We tested five master's degree students as those in experiment 3. We provided the source code of the sample tool that supports decision coverage and then explained the existing decision coverage to them in 15 min. Table 3 lists the number of people who extended the tool for Java to support the special decision coverage within 180 min. The average time required to extend the tool for Java using OCCF was 34.2 min. The examinees attempted to modify Cobertura to measure the special decision coverage. However, nobody completed this task within 180 min.

Experiment 5: We carried out an experiment on the implementation of a new coverage criterion for Python that was a special statement coverage limited to print statements. Moreover, we investigated the maintenance required for an upgrade from Python version 2 to Python version 3 because Python version 3 does not have backward compatibility to version 2. We evaluated the cost of developing a new coverage criterion and the cost of the maintenance required to change the language specifications.

We tested four first year master's degree students studying computer science, who were not the examinees in experiment 4. We explained our framework to them in 30 min and then provided them with the AST-generation component for Python 3, which we implemented for them in 25 min. Table 3 lists the number of people who implemented a tool for Python 2 and responded to the upgrade to Python version 3 within a total of 240 min. The average time required to implement a tool for Python 2 using the OCCF was 13.5 min, and the time to respond to the upgrade to Python 3 using the OCCF was 47.5 min. The examinees attempted to modify SCP to measure the print statement coverage. However, nobody completed this task within 240 min.

The reason why we gained the above results in experiments 4 and 5 is that the numerous tools that exist are not highly modularized, making it difficult to find the part of the code that has to be modified in order to extend the tool. OCCF, on the other hand, is highly modularized and the examinees could easily extend and update tools, i.e., they only implemented and modified the ISelector interface in experiments 4 and 5.

For the standard tools, both the syntax analyzer and the semantics analyzer have to be maintained. However, only the code insertion subsystem has to be maintained in tools using OCCF. The AST-generation component can be updated by using existing software. The code-generation component does not need to be changed because the tokens there are not memorized in the AST are not often changed. The inserter of the AST-operation component also does not need to be changed because the inserter does not depend on specific languages. Moreover, the selector, generator, and tagger of the AST-operation component do not need to be changed as long as the syntax corresponding to the semantics that are focused on does not change. As OCCF only focuses on the syntax, only limited maintenance is required.

For example, when Java is upgraded from 1.4 to 5.0, OCCF would only be required to locate the enhanced for statement. `Cobertura` and `EMMA`, on the other hand, would also be required to respond to the generic types. Moreover, experiments 4 and 5 indicate that it is easier to modify tools using OCCF than with `Cobertura` and `SCP`.

Therefore, we succeeded in alleviating the problem (P_2) of the high cost of maintenance for a given language and AST that the four components can easily operate.

6.3 Consistency in Measurement

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

We measured the coverage for the software that was developed in Java and Python as an example. When the software was tested using integration testing, the coverage was measured by using `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 or the same statement coverage was measured. Therefore, there is a possibility that only coverage that is ineffective can be obtained as an indicator of the software testing.

However, OCCF could measure the coverage with the same criterion, such as the decision coverage, for all languages. Thus, the effective coverage as an indicator of the software testing could be obtained.

`Gcov` and `NCover` can also measure the coverage for many languages. However, `gcov` only runs under `GCC`, e.g., it does not run under Visual C++ [10]. It is difficult to add new languages because `gcov` requires users to implement compilers in `GCC`. `NCover` only supports languages that run under the .NET Framework.

OCCF can measure the coverage in any environment where the 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 the inconsistency in measurement.

6.4 Flexibility in Measurement

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

The existing tools do 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 the measurement elements such as the lines, blocks, methods, and classes. `Cobertura` could change the measurement elements by using regular expressions.

OCCF, on the other hand, could freely change the measurement range and elements based on the conditions set by XPath and the customized coverage criteria. For example, OCCF could limit the measurement range to the methods that contained ‘while’ statements based on the conditions set by XPath. Moreover, OCCF could limit the measurement elements to the statements that called on a specific method set by the customized coverage criteria in order to measure the special statement coverage that were limited to only the statements that called a specific method. This coverage could be used in library testing. Moreover, we confirmed that developers can add new coverage criteria in the experiment 4 and 5.

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

6.5 Completeness in Measurement

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

`Cobertura` inserts the measurement code into a Java bytecode. However, it does not measure the coverage of dead code because the compiler optimization facility removes the dead code from the bytecode.

However, OCCF inserts the measurement code into the source code before the compiler optimization facility removes the dead code. Therefore, OCCF can be detected at a part where the dead code has not been tested because the information that was inserted there remains. For instance, OCCF had 50% statement coverage as shown in Fig. 2 in Sect. 3.

Note that exception handlers are not dead code because they are not always executed. Both the existing tools and OCCF measure coverage for exception handlers. However, sometimes developers want to ignore exception handlers so that they obtain 100% coverage as already mentioned in Sect. 3.3 by executing only all statements except for exception handlers. OCCF can exclude exception handlers from the measurement elements by adding an exclusion condition described by XPath to user code.

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

6.6 Time Efficiency

We evaluated the time efficiency by using the time to execute three Java programs presented in a book [20]. This evaluation provided good results compared with the existing tools although we are not referring to the problem corresponding to this evaluation.

Experiment 6: We measured the time to execute three Java programs presented in a book. Measuring the coverage de-

Table 4 Execution time on the millisecond time scale during software testing.

	original code	Cobertura	OCCF (TCP/IP)	OCCF (file)
Huffman	60	2473	317	17691
Hash	17	283	28	8910
QuickSort	1	194	14	5350

creased 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 a TCP/IP communication compared with the former source code, as shown in Table 4. OCCF is about 10 times faster than `Cobertura`.

However, it is more than 1000 times slower than the original source code when used with the file output, and it is 10 to 30 times slower than `Cobertura`. The TCP/IP communication is overwhelmingly faster than a simple file output.

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

7. Related Work

Kiri et al. [21] and Rajan et al. [22] among others had similar ideas and we will now refer to their study results, and our approach bares a resemblance to the following existing techniques.

Kiri et al. proposed the idea of developing a tool that inserts measurement code into a source code. Their idea was to measure the statement coverage, decision coverage, and a special coverage called RC0. RC0 is a special statement coverage for only revised statements. However, their idea was to measure only the statement coverage and decision coverage because they measured the 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, it can easily support new coverages like RC0 by adding a user code.

Rajan et al. proposed the idea of specifying the measuring elements using a description style of pointcut that is used in Aspect-oriented programming languages. They demonstrated a tool that supported C#. Measuring the elements, such as the method calls, ‘if’ statements, exception handlers, and variable writes could be specified. However, the description style that specified the measurement elements was specialized for only 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.

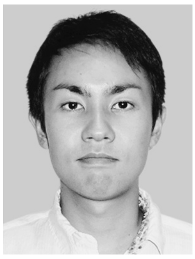
8. Conclusion and Future Work

We proposed OCCF, reduced development and maintenance costs, made flexible measurement, and made complete measurements by extracting the commonalities from multiple languages using an AST.

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 by using a wizard and the required user input through the GUI to further reduce the developmental costs. We believe OCCF can contribute to the development of the implementation of a processing system that transforms source code such as a refactoring tool into language-independent models.

References

[1] Lee Copeland, A Practitioner’s Guide to Software Test Design, Artech House, 2003.
[2] K. Sakamoto, Open Code Coverage Framework, http://sourceforge.jp/projects/codecoverage/
[3] K. Sakamoto, H. Washizaki, and Y. Fukazawa, “Reporting the implementation of a framework for measuring test coverage on design pattern,” Software Patterns and Quality (SPaQu 2009), 2009.
[4] K. Sakamoto, H. Washizaki, and Y. Fukazawa, “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] Microsoft, Visual C++, http://msdn.microsoft.com/visualc/
[11] K. Beck, Test-driven development: By example, The Addison-Wesley Signature Series, 2003.
[12] Y. Sakata, K. Yokoyama, H. Washizaki, and Y. 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.
[13] M. Fayad and D.C. Schmidt, “Object-oriented application frameworks,” Commun. ACM, vol.40, no.10, pp.32–38, Oct. 1997.
[14] Microsoft, Dynamic Language Runtime, http://dlr.codeplex.com/
[15] Microsoft, Managed Extensibility Framework, http://www.codeplex.com/MEF/
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
[17] OMG, Unified Modeling Language (UML) specification, version 2.2, http://www.omg.org/spec/UML/
[18] ANTLR, http://www.antlr.org/
[19] Apache Software Foundation, The Byte Code Engineering Library, http://jakarta.apache.org/bcel/
[20] H. Okumura, H. Satoh, K. Turu, K. Shudo, and T. Nobuyuki, “Algorithm cyclopedia by Java,” Gijutuhyoronsya, 2003.
[21] T. Kiri, T. Miyoshi, S. Kishigami, T. Osato, and T. Sonehara, “About the source code insertion type coverage tool,” 69th Information Processing Society of Japan National Convention, 2003.
[22] H. Rajan and K. Sullivan, “Aspect language features for concern coverage profiling,” International Conference on Aspect-Oriented Software Development (AOSD’05), pp.181–191, 2005.
[23] Software Freedom Conservancy, http://www.swig.org/
[24] K. Sakamoto, H. Washizaki, and Y. Fukazawa, “Open code coverage framework: A consistent and flexible framework for measuring test coverage supporting multiple programming languages,” The 10th International Conference on Quality Software (QSIC 2010), 2010.



Kazunori Sakamoto was born in 1987. He received the B.E. and M.E. degrees in Information and Computer Science from Waseda University, Tokyo, Japan in 2009 and 2010, respectively. He is now a doctoral student of Department of Information and Computer Science, Waseda University. His research interests include software engineering especially software testing.

Fuyuki Ishikawa received his Ph.D. degree in Information Science and Technology from The University of Tokyo, Japan, in 2007. Since April 2007, he has been an assistant professor at Digital Content and Media Sciences Research Division, National Institute of Informatics, Japan. He is also an assistant professor at Department of Informatics, School of Multidisciplinary Sciences, The Graduate University for Advanced Studies (Sokendai University), Japan. His research interests include Services Computing and Software Engineering.

Hironori Washizaki is an associate professor at Waseda University, Tokyo, Japan. He is also a visiting associate professor at National Institute of Informatics, Tokyo, Japan. He obtained his Doctor's degree in Information and Computer Science from Waseda University in 2003. His research interests include software reuse, patterns and quality assurance. He has served as members of program committee for many international conferences including ASE, SEKE, PROFES, APSEC and PLoP. He has also served as members of editorial board for several journals including Journal of Information Processing.

Yoshiaki Fukazawa received the B.E, M.E. and D.E. degrees in electrical engineering from Waseda University, Tokyo, Japan in 1976, 1978 and 1986, respectively. He is now a professor of Department of Information and Computer Science, Waseda University. Also he is Director, Institute of Open Source Software, Waseda University. His research interests include software engineering especially reuse of object-oriented software and agent-based software.