A Study on Source Code Reengineering Frameworks Supporting Multiple Programming Languages

複数のプログラミング言語に対応するソースコードリエンジニアリングフレームワークに関する研究

by

Kazunori Sakamoto

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Engineering

Graduate School of Science and Engineering
Waseda University

March 2013

A Study on Source Code Reengineering Frameworks Supporting Multiple Programming Languages

by

Kazunori Sakamoto

Submitted to the Department of Computer Science and Engineering
on February 18, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Engineering

Abstract

Program source code reengineering is one of key technologies in software development for improving software quality with low costs. For example, software metrics are important indicators for assessing software quality which are acquired by analyzing source code, test coverage is also an important indicator for assessing test quality which is acquired by transforming source code to store execution logs and an AOP (aspect-oriented programming) processor makes source code high modularity by weaving aspects into source code. Such means are based on source code reengineering and many kinds of reengineering tools exist.

Programming languages have become more diversified. There are many paradigms of programming languages: statically-typed / dynamically-typed, imperative / declarative, functional / logic and object-oriented / aspect-oriented / context-oriented. Software development using multiple programming languages is required because each programming language has an area of specialty and programmers choose suitable programming languages for each project. In particular, web applications are based on the client-server model and usually use three programming languages: HTML, JavaScript and another programming language. The development of web applications has been more and more frequent with the popularization of the Internet. The tools therefore are required to support multiple programming languages for such development.

However, there are two problems in existing tools. (1) It requires high costs to develop tools supporting multiple programming languages due to the variety of programming languages. Many tools thus support few programming languages and the users receive only little benefits from the tools. (2) There are differences between tools supporting different programming languages because each tool is developed for specific programming languages and supports few programming languages. For example, measurement tools of test coverage, Cobertura supports only Java and Coverage.py supports only Python. However, there is no tool which measures test coverage for TypeScript, thus, programmers cannot measure test coverage for TypeScript. Moreover, whereas Cobertura supports statement coverage and decision coverage, Statement coverage for Python supports only statement coverage. It is difficult to measure decision coverage for web applications using Java and Python. Therefore, means are required to reduce development costs of tools supporting multiple programming languages and to reduce inconsistency between tools.

To solve the problems, this thesis proposes a novel two frameworks called OCCF (Open Code Coverage Framework) and UNICOEN (Unified Code Reengineering Framework). OCCF is a consistent, flexible and complete framework for measuring test coverage supporting multiple programming languages. UNICOEN is a framework developed by generalizing OCCF for reengineering source code supporting multiple programming languages.

The thesis consists of 6 chapters.

Chapter 1 “Introduction” states the objective this research with research background. This chapter also defines the research area of the thesis by referring related works.

Chapter 2 “Software Development Using Multiple Programming Languages” describes the situation where software development using multiple programming languages is required. This chapter summarizes programming language paradigms and shows the relation of application types and suitable programming languages. The chapter also describes the importance of researches to overcome the variety of programming languages.

Chapter 3 “OCCF (Open Code Coverage Framework): A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages” describes the proposed framework called OCCF. Existing measurement tools have four problems. (1) Developing measurement tools supporting multiple programming languages requires high costs. There is no free measurement tool for legacy and new programming languages such as COBOL and Kotlin. Such situation makes it difficult to maintain and introduce legacy and new programming languages. (2) Existing tools, which support different programming languages, measure also different coverage criteria. The difference makes it difficult to introduce test coverage for software using multiple programming languages such as web applications. (3) It is difficult customize existing measurement tools for utilizing special coverage criteria. The effort to add new coverage criteria in the tools is significant. (4) Some existing tools insert measurement code into compiled binary file to measure test coverage. However, this way misses dead code because the compiler optimization removes any dead code. Users therefore cannot notice dead code from the result of test coverage. To overcome the problems, OCCF reduces development costs of measurement tools by providing reusable architecture and code. The reusable architecture and code help users to implement tools supporting consistent coverage criteria: statement coverage, decision coverage, condition coverage, condition/decision coverage. OCCF also provides hot spots to customize coverage criteria. Moreover, OCCF inserts the measurement code into source code and allows user to notice the existence of dead code. The effectiveness of OCCF is evaluated by experiments in the chapter. As a result of the experiments and the application, OCCF alleviates problems.

Chapter 4 “An Application of OCCF for Minimizing Test Cases Based on Test Coverage” describes a tool developed with OCCF. The tool minimizes test cases by judging whether a test case is duplicated with other test cases based on test coverage.

The tool considers that the test case which executes the same elements of production code executed by other test cases is duplicated in terms of suitable test coverage to measure such elements. This chapter also describes the evaluation of the tool through applications in existing open source software. As a result of the application development, this chapter shows the effectiveness of OCCF.

Chapter 5 “UNICOEN (Unified Code Reengineering Framework): UNICOEN: A Unified Framework for Code Reengineering Supporting Multiple Programming Languages” describes the proposed framework called UNICOEN. UNICOEN generalizes OCCF to support not only tools for measuring test coverage and also tools for analyzing and transforming source code. Existing tools have two problems. (1) Although some previous works propose frameworks for developing tools which reengineers source code, there is no framework which provides common language model with API of both analysis and transform and which allows users to extend supports of programming languages. Therefore, some programming languages have no tool for analyzing or transforming. For example, Lint, JSLint and Pyklint are static analyzers for finding bugs. They support C, JavaScript and Python respectively. However, no tool supports Ruby. (2) Many programmers use many programming languages. In particular, most developers of open source software use more than three programming languages. However, there are differences between existing tools, which supports different programming languages. For example, AspectJ supporting Java and AOJS supporting JavaScript are AOP language processors. AspectJ and AOJS support different join points and provide different grammars for writing aspects. Users therefore should learn the both grammars to introduce AOP in development of web applications using Java and JavaScript. To solve the problems, UNICOEN provides common language model, called UCM (Unified Code Model). UCM is developed by adding elements of seven programming languages: C, C#, Java, Visual Basic, JavaScript, Python and Ruby. UCM thus can represent source code of the seven programming languages in common model similarly. UNICOEN defines UCM in terms of syntaxes. UNICOEN assumes that similar syntax have similar semantics between different programming languages. Although UNICOEN cannot interpret semantics of all elements from syntax completely, most elements can be distinguished and can be interpreted because the assumption is valid for most cases. This feature dramatically reduces costs for adding new supports of programming languages compared to other frameworks which interpret semantics. UNICOEN provides two kinds of API for adding supports of programming languages in UNICOEN and for developing tools for reengineering source code. The API provides reusable code and useful methods similar to LINQ in .NET framework. UNICOEN therefore reduces costs for developing tools by providing such API for reengineering source code. The chapter also describes measurement tool of cyclomatic complexity with UNICOEN for evaluating UNICOEN. In comparison to a similar tool for Ruby, the tool with UNICOEN has less lines of code and supports more programming languages. Moreover, the chapter describes evaluations in comparison to programming languages processors and other frameworks. The evaluations indicate that UNICOEN reduces costs to add new programming language supports and to develop tools supporting multiple programming languages. As a result, OCCF alleviates problems.

Chapter 6 “An Application of UNICOEN for Aspect-Oriented Programming Processors” describes a tool developed with UNICOEN. The tool is an AOP processor supporting seven programming languages supported by UNICOEN, called UniAspect. UniAspect provides four join points for supported programming languages similarly: call, execution, get and set. UniAspect also provides aspects which contain language-independent pointcuts and a set of advices written in each programming languages to weave. UniAspect language-independently weaves the aspects into source code by modifying objects on UCM. As a result of UniAspect development, this chapter shows the effectiveness of UNICOEN.

Chapter 7 “Conclusions” concludes the thesis and explains future works.

Acknowledgments

To finish the work presented in this thesis, I owe much to a lot of persons. First, I would like to thank Assoc. Prof. Hironori Washizaki for his supervisor and support on my research. This thesis was accomplished with the help of Prof. Yoshiaki Fukazawa, Prof. Kazunori Ueda, Assoc. Prof. Kazuyuki Shudo as sub-examiners, and other professors of the Department of Computer Science and Engineering, Waseda University.

I also show sincere thanks to Assoc. Prof. Takashi Kobayashi (Department of Computer Science, Tokyo Institute of Technology), Prof. Katsuhiko Kakehi (Department of Computer Science and Engineering, Waseda University), Dr. Atsuto Kubo, Mr. Daigo Hamura (Google Japan Inc.), Mr. Kaizu Tomohiro (Google Japan Inc.), Mr. Daichi Ota (ACCESS CO., LTD.), Mr. Akira Ohashi (Sony Corporation), Mr. Iwasawa Hiroki (Yahoo Japan Corporation) and Mr. Kazuki Eguchi (Yahoo Japan Corporation). Plenty of discussions and advice have simulated and deepened my research analysis.

A part of this thesis was accomplished with the help of the member in Washizaki Laboratory, Fukazawa Laboratory and Ueda Laboratory, Waseda University. I would like to thank Mr. Satoru Uchiyama, Mr. Yuki Shiroma, Mr. Yutaro Nomoto, Mr. Akihiko Syoyama, Mr. Yuto Nakamura, Mr. Masaya Shimizu, Mr. Syuhei Takahashi, Mr. Shinichi Murakami, Ms. Reisha Humaria, Mr. Kiyofumi Shimojo, Mr. Ryushi Shiohama, Mr. Tomoyuki Kamiya, Mr. Masahiko Wada, Mr. Satoshi Arai, Mr. Shohei Aoi, Mr. Seiji Sato, Mr. Fumiya Kato, Mr. Ryohei Takasawa, Mr. Koichi Takano and Mr. Naoki Yaguchi. I am also thankful to all staffs and members in Washizaki Laboratory and Fukazawa Laboratory.

This thesis has been partially supported by the funds of IPA Mitou-youth project, ROHM Co., Ltd., Benesse Corporation and The Foundation of Ando Laboratory. Also, many (anonymous) referees contributed to the improvement of this thesis via conferences and paper submissions.

Finally, I would like to thank my wife, Keiko, and my family for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Program-source-code reengineering, which is transforming source code by analyzing the source code, is one of key technologies in software development for improving software quality with low costs. Source-code reengineering provides various techniques such as measuring test coverage [86], measuring software metrics [23, 52, 55] and aspect-oriented programming (AOP) [20, 46, 48]. Test coverage is an important indicator for assessing test quality which is acquired by transforming source code to store execution logs. Software metrics are also important indicators for assessing software quality which are acquired mainly by analyzing source code. AOP makes source code high modularity by weaving aspects into source code to achieve separation Of concerns. To put these techniques into practical use, many kinds of reengineering tools were developed.

Source-code reengineering treats source code, which can be written in various programming languages. The diversity of programming languages makes the structure and contests of source code different. These differences cause the differences between tools for reengineering source code. Consequently, the differences cause important problems.

In this chapter, I describe the current problematic state of reengineering tools after explaining the current state of programming languages.

1.2 Programming languages

Programming languages have become so diversified that 2500 programming languages at least have been developed [50]. Tables 1.1 and 1.2 show the classification of major programming languages with the following aspects: whether procedural or declarative, whether statically typed or dynamically typed, whether object-oriented or not and whether functional or not. Note that JavaScript supports prototype instead of class and inheritance (* in Table 1.1) and C99 supports a pointer, which can be considered as a parameter and a return value, instead of a function (* in Table 1.2).

Most programming languages developed recently have functional features. For example, C++11, which was approved by ISO in 2011, have functional features such as lambda functions and expressions to create anonymous functions. Scala, which is being developed to replace Java, is a multi-paradigm programming language as it supports functional, object-oriented and imperative programming. Although C++11 and Scala have new features, they have also traditional language features such as `if` statements. In this way, programming languages has been evolving maintaining these features.

As another trend of the times, numerous programming languages which work with other programming languages are being developed. In particular, most of new programming languages work with Java, .NET languages or JavaScript. Tables 1.3 and 1.4 show the lists of programming languages which work on Java virtual machines and .NET Framework, respectively. Note that Java can work on .NET Framework with IKVM.NET [1] which is an implementation of Java for .NET Framework (* in Table 1.3), Java is partially convertible to JavaScript with Google Web Toolkit [2] (** in Table 1.3) and C# is partially convertible to JavaScript with converters such as SharpKit [3] and Script# [4] (* in Table 1.4. Java and .NET languages such as C# work on virtual machines beyond operating systems. The virtual machines are developed with Java and .NET languages. However, other programming languages are developed

[1]http://www.ikvm.net/
[2]https://developers.google.com/web-toolkit/
[3]http://sharpkit.net/
[4]http://scriptsharp.com/

for the virtual machines such that they run on various environments as well as Java and .NET languages. Obviously, these programming languages are compatible with Java or .NET languages through virtual machines. For example, Scala, which works on the Java virtual machine, can invoke methods written in Java.

Table 1.5 shows the list of programming languages which are convertible to JavaScript. Note that Haxe is convertible to many other programming languages such as C++, PHP and C# (* in Table 1.5). JavaScript does not use virtual machines, while most web browsers support JavaScript. Because numerous web browsers work on various operating systems, JavaScrip also work on various operating systems. JavaScript is widely used for implementing client applications which have GUI due to spread of web applications and the growing popularity of smartphones and tablet PCs. However, JavaScript is prototype-based so that JavaScript is different with other popular programming languages such as C++ and Java. Thus, programming languages which are convertible to JavaScript are being developed. For example, CoffeeScript; which is inspired by Python, Ruby and Haskell; is convertible to JavaScript program. In this way, the programming languages in Tables 1.3, 1.4 and 1.5 aid developers to use multiple programming languages.

1.3 Current state of source code reengineering

This section describes the current state of source code reengineering highlighting test coverage.

1.3.1 Test coverage

Test coverage (also known as code coverage), which provides quantitative values for assessing test adequacy, has various criteria such as statement coverage, decision coverage, condition coverage, and condition/decision coverage. For example, statement coverage is the ratio of statements that have been executed at least once from all the statements. Based on the purpose of software testing, the developer selects the suitable criterion [14].

Table 1.1: Classification of programming languages in terms of paradigms

Programming language	Procedural / Declarative	Type system	Class and inheritance
Pascal	Procedural	Statically typed	No
COBOL85	Procedural	Statically typed	No
Fortan90	Procedural	Statically typed	No
C99	Procedural	Statically typed	No
C++11	Procedural	Statically typed	Yes
C# 4.0	Procedural	Statically typed	Yes
D 2.0	Procedural	Statically typed	Yes
Objective-C 2.0	Procedural	Statically typed	Yes
Java 7	Procedural	Statically typed	Yes
JavaScript	Procedural	Dynamically typed	Yes(*)
CoffeeScript	Procedural	Dynamically typed	Yes
TypeScript	Procedural	Statically typed	Yes
Perl	Procedural	Dynamically typed	Yes
PHP 5.3	Procedural	Dynamically typed	Yes
Python	Procedural	Dynamically typed	Yes
Ruby	Procedural	Dynamically typed	Yes
Lua	Procedural	Dynamically typed	No
Groovy	Procedural	Dynamically typed	Yes
Lisp	Procedural	Dynamically typed	No
Scheme	Procedural	Dynamically typed	No
ML	Procedural	Statically typed	No
Scala	Procedural	Statically typed	Yes
Kotlin	Procedural	Statically typed	Yes
Nemerle	Procedural	Statically typed	Yes
Haskell	Declarative	Statically typed	No
SQL	Declarative	Statically typed	No
Prolog	Declarative	Dynamically typed	No

Table 1.2: Classification of programming languages in terms of functional features

Programming language	Function as parameter	Function as return value	Closure
Pascal	No	Yes	No
COBOL85	No	No	No
Fortan90	Yes	Yes	No
C99	Yes(*)	Yes(*)	No
C++11	Yes	Yes	Yes
C# 4.0	Yes	Yes	Yes
D 2.0	Yes	Yes	Yes
Objective-C 2.0	Yes	Yes	Yes
Java 7	No	No	No
JavaScript	Yes	Yes	Yes
CoffeeScript	Yes	Yes	Yes
TypeScript	Yes	Yes	Yes
Perl	Yes	Yes	Yes
PHP 5.3	Yes	Yes	Yes
Python	Yes	Yes	Yes
Ruby	Yes	Yes	Yes
Lua	Yes	Yes	Yes
Groovy	Yes	Yes	Yes
Lisp	Yes	Yes	Yes
Scheme	Yes	Yes	Yes
ML	Yes	Yes	Yes
Scala	Yes	Yes	Yes
Kotlin	Yes	Yes	Yes
Nemerle	Yes	Yes	Yes
Haskell	Yes	Yes	Yes
SQL	No	No	No
Prolog	No	No	No

Table 1.3: Classification of programming languages which can work with other languages (mainly Java)

Programming language	on JVM	on .NET Framework	convertible to JavaScript	Other compatible languages
Java	Yes	Yes(*)	Yes(**)	
Scala	Yes	Yes	No	
Kotlin	Yes	Planned	Yes	
Xtend	Yes	No	No	
Ceylon	Yes	No	Yes	
Fantom	Yes	Yes	Yes	
Gosu	Yes	No	No	
Clojure	Yes	No	No	Lisp
Jython	Yes	No	No	Python
JRuby	Yes	No	No	Ruby

Table 1.4: Classification of programming languages which can work with other languages (mainly .NET)

Programming language	on JVM	on .NET Framework	convertible to JavaScript	Other compatible languages
C#	No	Yes	Yes(*)	
F#	No	Yes	No	
Visual Basic	No	Yes	No	
C++/CLI	No	Yes	No	
Nemerle	No	Yes	Yes	
IronPython	No	Yes	No	Python
IronRuby	No	Yes	No	Ruby
IronScheme	No	Yes	No	Scheme
Produire	No	Yes	No	
Kurogane	No	Yes	No	

Table 1.5: Classification of programming languages which can work with other languages (mainly JavaScript)

Programming language	on JVM	on .NET Framework	convertible to JavaScript	Other compatible languages
CoffeeScript	No	No	Yes	
TypeScript	No	No	Yes	
Dart	No	No	Yes	
Haxe(*)	Yes	Yes	Yes	ActionScript
JSX	No	No	Yes	

Although there are many programming languages and coverage criteria, most coverage-measurement tools support limited programming languages and coverage criteria. Consequently, many tools exist to span the various programming languages, which leads to differences between existing tools [83, 84]. These differences prevent testers from accurately measuring coverage because the tools support different coverage criteria and are implemented with different ways, resulting in different values for the same criteria.

For example, EMMA [69] supports statement coverage for Java, while Coverage.py supports both statement coverage and decision coverage for Python. Although a combination of EMMA and Coverage.py can measure statement coverage of a web application using Java and Python, the combination cannot measure decision coverage. Moreover, EMMA divides a ternary expression (`condition ? true-expression : false-expression`) into two statements and can determine whether both branches of the ternary expression have been executed. On the other hand, Coverage.py cannot determine whether both branches have been executed because it does not divide ternary expressions.

Tables 1.6, 1.7 and 1.8 compare existing coverage-measurement tools by features and supported programming languages. An 'X' in the tables indicates the tool has the feature to support a criterion or programming language. 'Binary' and 'Code' indicate the way to modify compiled binary file or source code to instrument code, respectively. 'Processor' indicates the way to monitor execution logs with a processor.

'Compiler' indicates the way to insert instrument code with a compiler.

As the tables show, there are many tools. These tools support different programming languages owing to their implementations. To the best of my knowledge, no free tool supports the four coverage criteria. Few tools support the scripting languages such as JavaScript, Python, Ruby and Lua. Consequently, these languages have poor features and support few coverage criteria. Although undercover is a free tool, which supports more than one programming language, undercover targets only programming languages working on Java Virtual Machine (Java VM). Non-free tools, except Clover and Semantic Designs, support only C/C++, C# and Java because these programming languages are most common. Therefore, accurately measuring coverage for web applications using scripting languages is difficult.

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

1.3.2 Software metrics

Software metrics provide quantitative values for assessing software quality. Software has various aspects and properties, and thus various software metrics are required to measure them. For example, McCabe proposed a complexity metric for general programming languages [59]. Chidamber and Kemerer, Li and Henry also proposed metrics suites for object-oriented programming languages [10, 11, 56].

As well as test coverage, there are many metrics measurement tools so that the diversity of the tools causes problematic differences. Lincke et al. [57] studied metrics measurement tools and found existing tools result different measurement values. Moreover, they calculated maintainability by combining measurement values of OOP metrics. As a result, they also found the differences in existing tools led to different conclusions by comparing the ranking of the classes according maintainability.

Table 1.6: Comparison of coverage-measurement tools in terms of features

Tool	Instrumentation	Minimal targets	Free
Cobertura	Binary	Function	Yes
EMMA	Binary	Function	Yes
JCover	Code	Function	No
Clover	Code	Function	No
Agitar	Binary	Function	No
OpenCover	Processor	Function	Yes
NCover	Source	Function	No
dotCover	Binary	Function	No
gcov	Compiler	File	Yes
COVTOOL	Code	File	Yes
BullseyeCoverage	Code	Function	No
Intel Code Coverage Tool	Compiler	Function	No
Squish Coco	Code	Function	No
TCAT	Code	Function	No
Parasoft Test	Code	Function	No
PurifyPlus	Binary	Function	No
Semantic Designs	Code	Function	No
CoverageValidator	Code	Function	No
ScriptCover	Code	File	Yes
Coverage.py	Processor	Module	Yes
rcov	Processor	Function	Yes
SimpleCov	Processor	Function	Yes
Devel::Cover	Processor	Function	Yes
xdebug	Code	Function	Yes
LuaCov	Processor	File	Yes

Table 1.7: Comparison of coverage-measurement tools in terms of coverage criteria

Tool	Coverage criteria			
	Statement	Dicision	Condition	Condition/Decision
Cobertura	Yes	Yes	No	No
EMMA	Yes	Yes	No	No
JCover	Yes	Yes	No	No
Clover	Yes	Yes	No	No
Agitar	Yes	Yes	No	No
OpenCover	Yes	Yes	No	No
NCover	Yes	Yes	No	No
dotCover	Yes	No	No	No
gcov	Yes	Yes	No	No
COVTOOL	Yes	No	No	No
BullseyeCoverage	Yes	Yes	Yes	Yes
Intel Code Coverage Tool	Yes	No	No	No
Squish Coco	Yes	Yes	Yes	Yes
TCAT	No	Yes	No	No
Parasoft Test	Yes	Yes	Yes	Yes
PurifyPlus	Yes	Yes	Yes	Yes
Semantic Designs	Yes	Yes	No	No
CoverageValidator	Yes	Yes	No	No
ScriptCover	Yes	No	No	No
Coverage.py	Yes	Yes	No	No
rcov	Yes	No	No	No
SimpleCov	Yes	No	No	No
Devel::Cover	Yes	Yes	Yes	No
xdebug	Yes	No	No	No
LuaCov	Yes	No	No	No

Table 1.8: Comparison of coverage-measurement tools in terms of supported programming languages

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

1.3.3 Aspect-oriented programming

AOP is a new programming paradigm using an aspect for modularizing cross-cutting concerns to achieve separation of concerns. AOP does not replace object-oriented programming (OOP) but supplements OOP. Thus, most AOP languages extend existing OOP languages. For example, AspectJ [49], AspectC++ [78], AOJS [80] and AspectR [9] are a AOP languages which extends Java, C++, JavaScript and Ruby, respectively.

There are AOP language processors as well as test coverage. Although each AOP language has similar features using same paradigm, they provides different grammars for expressing aspects. For example, AspectJ defines a grammar based on Java, while AOJS defines a grammar based on XML. Therefore, programmers should learn various grammars when using AOP for multiple programming languages.

Chapter 2

Software Development Using Multiple Programming Languages

Recently, software development using multiple programming languages has been increasing.

Jones [40] reported most applications used between 2 and 15 programming languages. He explained that the reasons were because there were most applications across multiple problem areas of software applications and because these problem areas were larger than solution areas of programming languages. He also listed 10 problem areas of critical software such as mathematical calculations. He claimed that most applications contain four areas at least and that most programming languages were optimized for between 1 and 3 areas. However, an increase in the number of using programming languages makes it difficult to develop applications. He described how both development and maintenance costs increased as the number of languages in an application increased. For example, development and maintenance costs increase by 13% and 20%, respectively, when the number of programming languages in an application is 4.

Karus et al. [44] studied 22 open source software (OSS) repositories over 12 years. They found developers worked with more than 4 different languages including make and XML in a project on average. They showed which programming languages were used together by each developer such as a C/C++ developer and a Java developer. For example, a total of 79%, 30% and 23% of Java developers worked on XML, JavaScript and Groovy files, respectively, while a total of 18%, 2% and 2% of C/C++ developers worked on XML, JavaScript and Groovy files, respectively. They also found distinct dependencies between languages or artifact types commonly edited together.

In the following sections, I describe three examples of software development using multiple programming languages explaining my experiences. The examples are classified as development of distributed system which components run on different environments, of software across different problem areas and of execution environment supporting multiple programming languages.

2.1 Distributed system which components run on different environments

Web applications, which are based on a client-server computing model, are representative of distributed systems. While the kinds of executable programming languages on client side (i.e. on web browsers) are very few such as JavaScript, ones on server side are not limited by execution environments and thus very various.

For example, I developed a web application to hold my programming contest in the orientation for the 2nd year degree students on Waseda University. The web application shows predefined problems which include problem statements, inputs and expected outputs. Users can submit output files which can be calculated from problem statements and inputs to solve problems. The web application only requires output files. Thus, users can solve problems in diverse ways such as writing programs to generate output files or writing output files by hand. I developed the web applications with Ruby on rails. The web application uses Ruby on server side and JavaScript on client side.

2.2 Applications across different problem areas

As previously stated, most applications exist across different problem areas although most programming languages focus on few problem areas. Thus, such applications require to use multiple programming languages.

For example, I developed a tool, called POGen, for generating skeleton test code with accessor methods for template variables, which are dynamic parts in HTML files. POGen has two features: analyzing HTML template files and generating test code, which exist in different problem areas. I developed both features in Java at first. However, Java has little support for building strings including test code. I re-implemented the generation feature in Xtend to improve the readability of my source code. Xtend supports template expressions that allow string literals containing expressions. Figure 2-1 shows sample Xtend code for generating a getter and a setter of Java.

```
def generateGetterAndSetter(String capitalizedName) '''
  public int get<<capitalizedName>>() {
    return <<capitalizedName>>;
  }
  public void set<<capitalizedName>>(int <<capitalizedName>>) {
    this.<<capitalizedName>> = <<capitalizedName>>;
  }
'''
```

Figure 2-1: Sample Xtend code for generating a getter and a setter

Another example is utilizing a script language for customizing software settings. I developed a fault localization tool with my framework for measuring coverage. My framework is implemented using C#, and thus, my fault localization tool is also implemented using C#. I also utilizes IronPython to customize metrics for calculating suspiciousness scores of statements. Although C# requires compiling, ironPython is a script language and does not require compiling. Thus, users can easily customize metrics by changing the IronPython script. Figure 2-2 shows an IronPython script for calculating suspiciousness scores using James et al.'s metrics [41].

```
def CalculateMetric(executedAndPassedCnt, passedCnt, executedAndFailedCnt, failedCnt):
  if passedCnt == 0 and failedCnt == 0:
    return [float("nan"), float("nan"), float("nan")]
  if passedCnt == 0:
    return [1.0, float("nan"), executedAndFailedCnt / failedCnt]
  if failedCnt == 0:
    return [0.0, executedAndPassedCnt / passedCnt, float("nan")]

  p = executedAndPassedCnt / passedCnt
  f = executedAndFailedCnt / failedCnt

  if (p + f) == 0:
    return [float("nan"), p, f]

  return [f / (p + f), p, f]
```

Figure 2-2: IronPython script for calculating suspiciousness scores using James et al.'s metrics

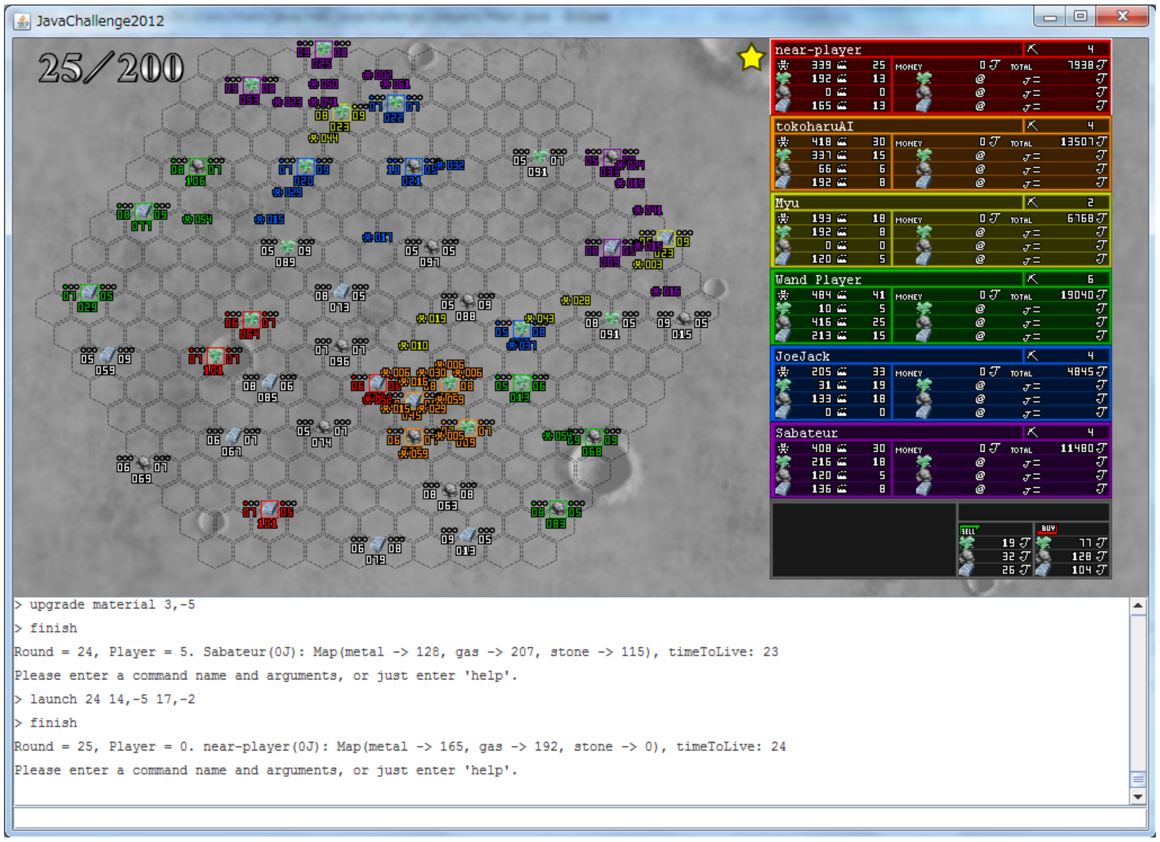
2.3 Execution environment supporting multiple programming languages

As a special case, I show an example of an execution environment for multiple programming languages. Such environment are usually developed using multiple programming languages.

I and my developer team developed platforms to hold programming contests with AI programs on my game software. JavaChallenge, which is held in conjunction with the ACM International Collegiate Programming Contest (ICPC), is one of famous programming contests using game AI programs. To hold JavaChallenge in conjunction with ICPC Asia Regional Contest 2012 in Tokyo, I and my developer team developed a game platform in Scala and Java to implement game software and to provide API for developing API. Figure 2-3 shows a screen shot of the game software for JavaChallenge 2012. My game, which was inspired by "The Settlers of Catan" [1] and "Galcon" [2], was based on turn-based strategy. I designed my game platform for game software to have the state of the game as an immutable object. Scala is suitable for implementing immutable objects rather than Java. However, JavaChallenge is a Java programming contest as the name suggests. Obviously, Java is suitable for

[1]http://www.catan.com/
[2]http://www.galcon.com/

providing Java API rather than Scala. In this way, I used both Java and Scala to develop my game platform.



```
> upgrade material 3,-5
> finish
Round = 24, Player = 5. Sabateur(0J): Map(metal -> 128, gas -> 207, stone -> 115), timeToLive: 23
Please enter a command name and arguments, or just enter 'help'.
> launch 24 14,-5 17,-2
> finish
Round = 25, Player = 0. near-player(0J): Map(metal -> 165, gas -> 192, stone -> 0), timeToLive: 24
Please enter a command name and arguments, or just enter 'help'.
```

Figure 2-3: Screen shot of Game Software for JavaChallenge 2012

Chapter 3

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

3.1 Introduction of this chapter

Test coverage or code coverage, which I 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

[14].

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 I 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.

I 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

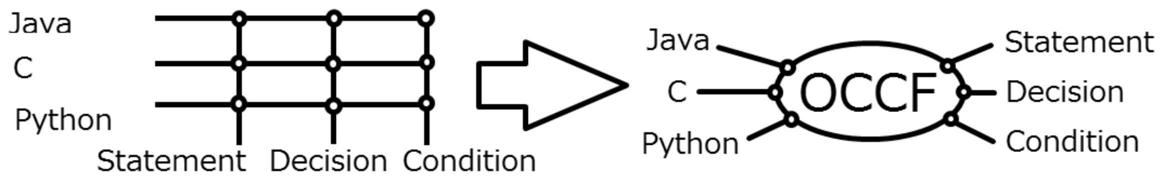


Figure 3-1: Simplification of connection between languages and coverage

an application-specific code to the framework [22].

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 3-1 outlines the concept behind the simplification, which is provided by OCCF. There are many-to-many relationships 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.

I 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, I 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 [71].

3.2 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 Statement coverage for Python (SCP) [2], which supports Python, and gcov, which supports the languages that the GNU Compiler Collection (GCC) [25] supports. The SCP uses a trace module in the Python standard library. Gcov is a subset of 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 [21], which supports Java, EMMA [69] for Java, and NCover [30] .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 my 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 ad-hoc 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.

3.3 Problems with conventional measurement approaches

3.3.1 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.

3.3.2 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.

3.3.3 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 my investigations, except for gcov and NCover, free tools that support multiple languages do not exist. I will discuss these points in Section 3.6. Therefore, consistent tools supporting multiple languages are required.

3.3.4 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 [4], 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. [73] 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 my 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).

3.3.5 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.

Figure 3-2 shows an example of a source code that includes dead code. 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");
    }
  }
}
```

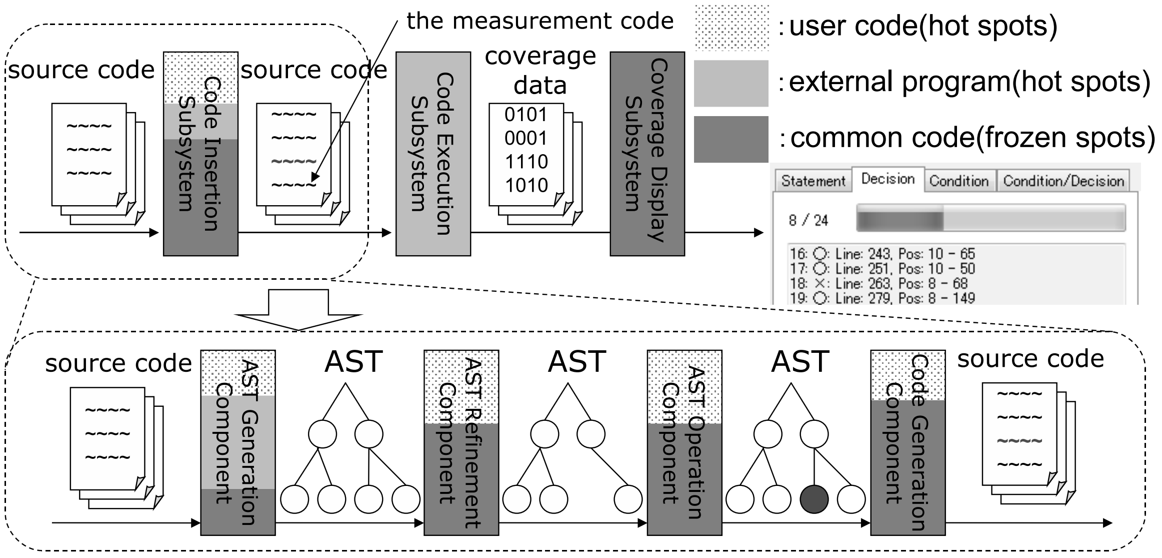
Figure 3-2: Sample code of dead code in Java

3.4 Framework for measuring coverage supporting multiple programming languages

I propose OCCF to support multiple languages, and which will alleviate the problems outlined in Section 3.3.

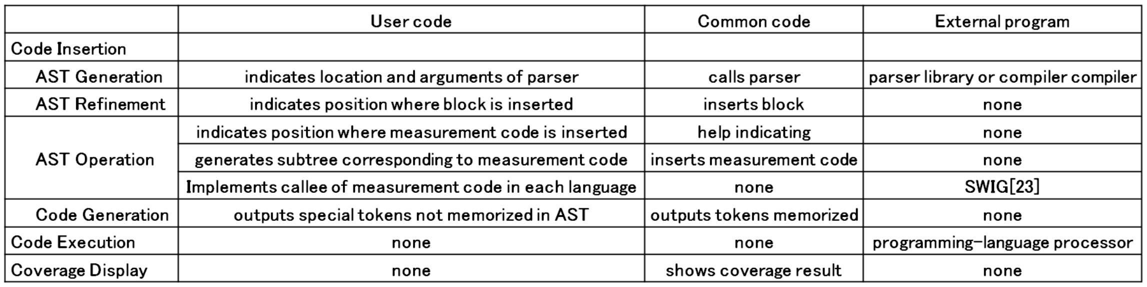
3.4.1 Measurement approach of OCCF

OCCF inserts a measurement code into the source code using AST, and the coverage is measured by executing the program. My approach analyzes the syntax and the required part of the semantics because my approach parses the source code to get the AST and locates the position of the node where the measurement code is inserted.



```
8 / 24
16: ○: Line: 243, Pos: 10 - 65
17: ○: Line: 251, Pos: 10 - 50
18: ×: Line: 263, Pos: 8 - 68
19: ○: Line: 279, Pos: 8 - 149
```

Figure 3-3: Overview of OCCF



	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

Figure 3-4: Overview of relation between user code, common code and external program

There is wide commonality in my approach among the measurement features because the insertion processing using the AST in each language is similar.

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

Figure 3-5: Sample code written in C prior to inserting the instrumentation code

The source code before it is inserted is outlined in Figure 3-5. The source code after being inserted is outlined in Figure 3-6. The `stmt_record` and `decision_record`

```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");
  }
}
```

Figure 3-6: Sample code written in C after inserting the instrumentation code

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.

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.

3.4.2 Overview of OCCF

An overview of OCCF and the processing flow in shown in Figure 3-3. 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
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 Figure 3-4. 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. I 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 $\mathtt{P}_4$.

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 $\mathtt{P}_5$.

3.5 Implementation of OCCF

I 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) [61]. The DLR is a .NET library that provides language services for several different dynamic languages. Moreover, OCCF uses the Managed Extensibility Framework (MEF) [62]. 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.

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

3.5.1 Code-insertion subsystem

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

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 [27].

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 [32] that is related to this component is shown in Figure 3-7. 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 `AntlrAstGenerator` calls the parser, which is generated by ANTLR [66], and outputs the AST as an XML document to help the users to use 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 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 Figure 3-8.

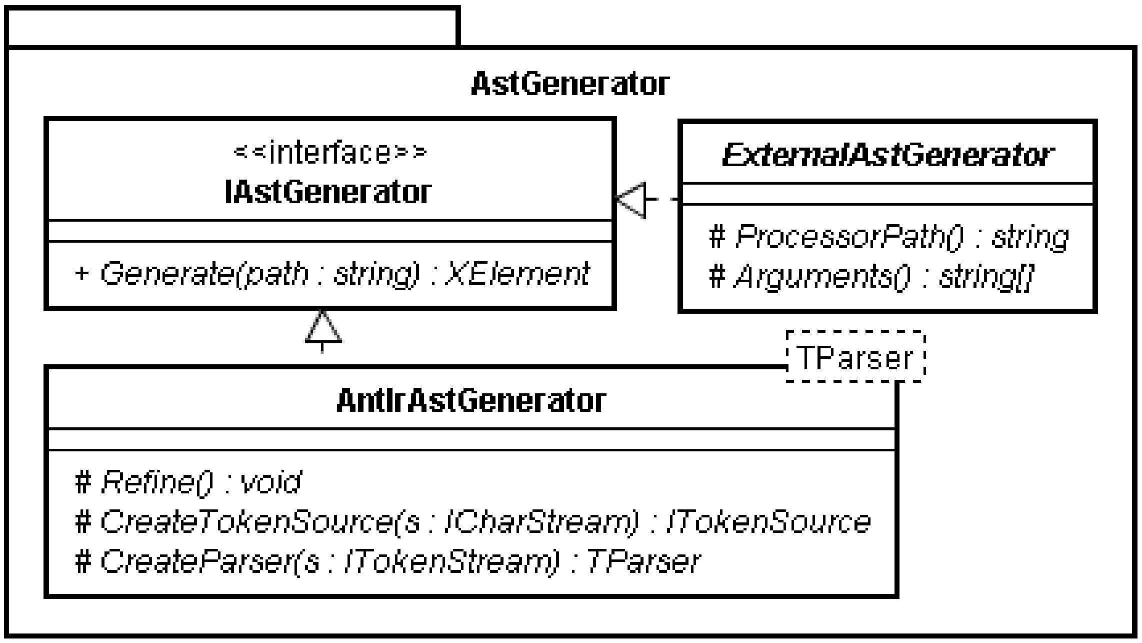

AstGenerator
<<interface>>
IAstGenerator
+ Generate(path : string) : XElement
ExternalAstGenerator
# ProcessorPath() : string
# Arguments() : string[]
TParser
AntlrAstGenerator
# Refine() : void
# CreateTokenSource(s : ICharStream) : ITokenSource
# CreateParser(s : ITokenStream) : TParser

Figure 3-7: 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;
    }
  }
}
```

Figure 3-8: PythonAstGenerator.cs

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 Figure 3-9.

```
[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)));
  }
}
```

Figure 3-9: CLackingBlockSelector.cs

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 Figure 3-10. 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 Figure 3-11 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 [27].

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.

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

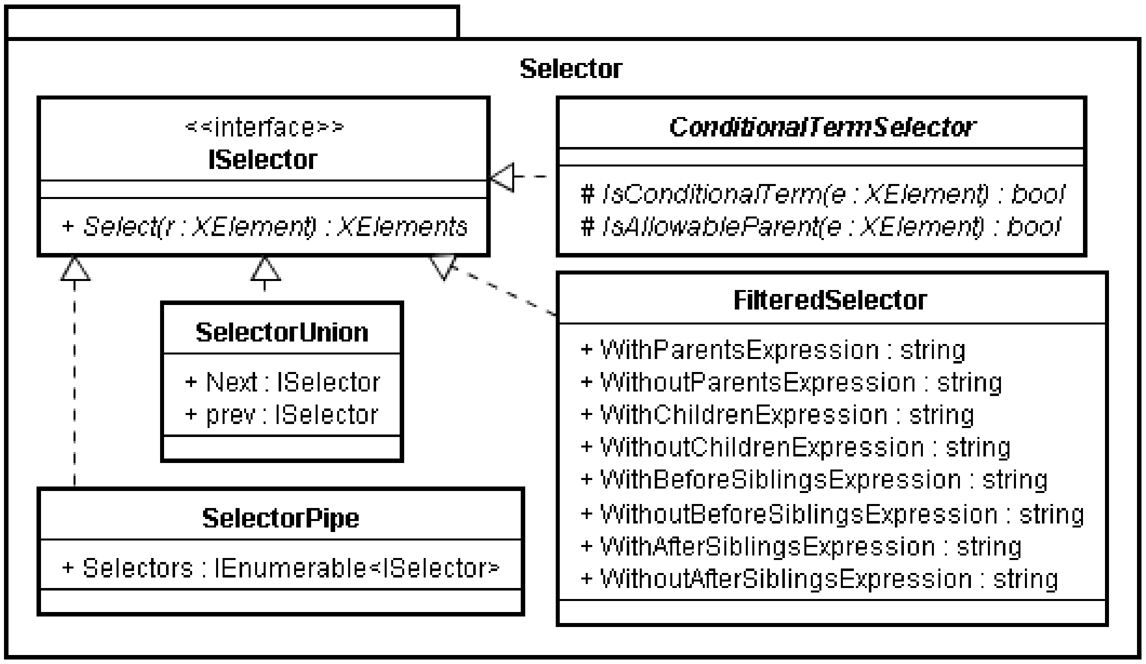

Selector
<<interface>>
ISelector
+ Select(r : XElement) : XElements
ConditionalTermSelector
# IsConditionalTerm(e : XElement) : bool
# IsAllowableParent(e : XElement) : bool
SelectorUnion
+ Next : ISelector
+ prev : ISelector
FilteredSelector
+ WithParentsExpression : string
+ WithoutParentsExpression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

Figure 3-10: Class diagram of selector in OCCF

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

```
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;
  }
}
```

Figure 3-11: ConditionalTermSelector.cs

`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 Figure 3-12.

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 [13] to implement it. Users only have to learn to use SWIG or manually

```
[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);
  }
}
```

Figure 3-12: PythonConditionalTermSelector.cs

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.

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 Figure 3-13. OCCF provides a `CodeGenerator` class that is designed by using the Template Method pattern and scans the AST and outputs the memorized tokens.

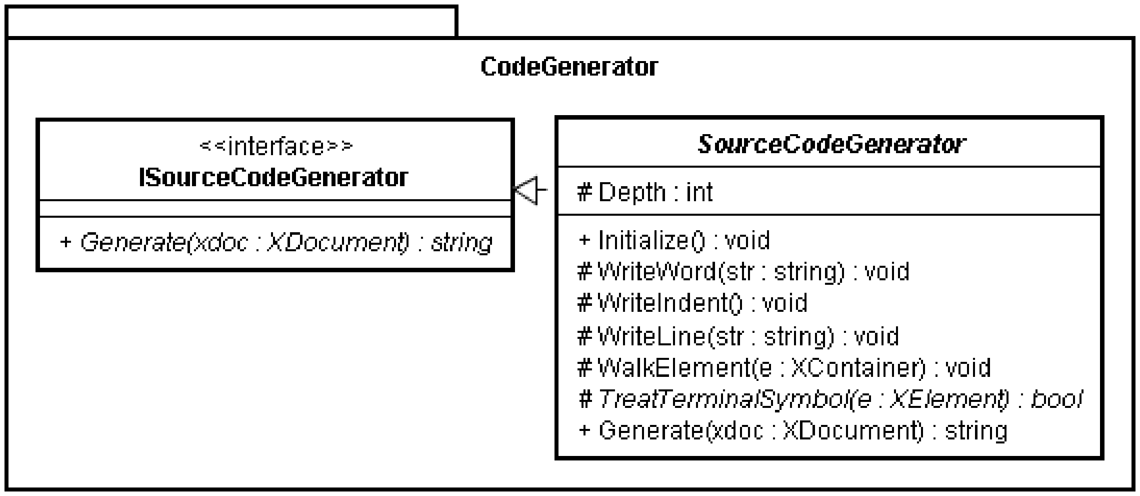

CodeGenerator
<<interface>>
ISourceCodeGenerator
+ Generate(xdoc : XDocument) : string
SourceCodeGenerator
# Depth : int
+ Initialize() : void
# WriteWord(str : string) : void
# WriteIndent() : void
# WriteLine(str : string) : void
# WalkElement(e : XContainer) : void
# TreatTerminalSymbol(e : XElement)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

Figure 3-13: Class diagram of code-generator component in OCCF

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 Figure 3-14.

```
[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;
    }
  }
}
```

Figure 3-14: PythonCodeGenerator.cs

3.5.2 Code-execution subsystem

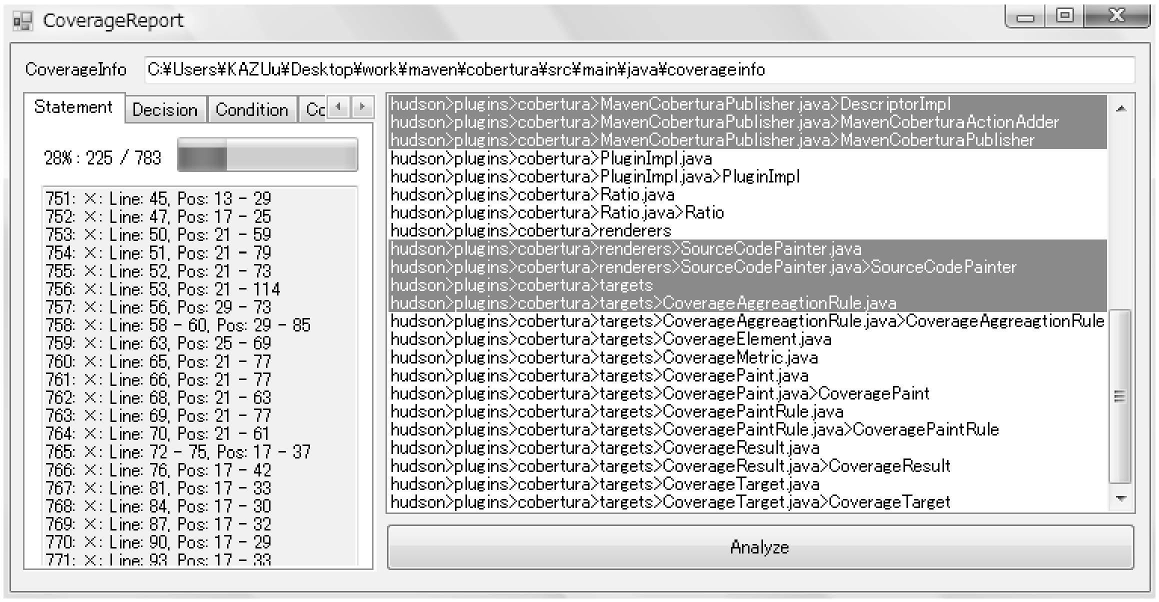
The code-execution subsystem executes the program in which the measurement code has been inserted. By executing the program, this subsystem sends the coverage information to the coverage-display subsystem. OCCF supports communications using TCP/IP, the shared memory, and the file output as the sending 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.

3.5.3 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 Figure 3-15. 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.



```
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
```

Figure 3-15: GUI Reporter in coverage-display subsystem

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.

Table 3.1: 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
OCCF:Completeness	yes	no	no	yes	no	yes
Non-commercial	yes	yes	yes	yes	yes	no

Table 3.2: 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)	-

3.6 Evaluation

I evaluated OCCF by comparing implemented samples that were developed by using OCCF with standard tools that are used as described in Section 3.3. 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 3.1 provides a comparison between OCCF and the other tools.

Experiment 1: I 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. I targeted three Java programs presented in a book [34] that use typical programming constructors and algorithms. I translated these Java programs into C and Python.

Table 3.2 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. I 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 3.2 indicates the tools can not measure condition coverage and.

The measurement results cannot be directly compared. There are three differences between OCCF and the state-of-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, I 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, I 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. I can think that abbreviated default constructors are not statements because abbreviated default constructors do not appear in source code. I 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. I 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, I confirmed that OCCF measures coverage accurately.

3.6.1 Reduced cost of new developments

I 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: I 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 Figure 3-16 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 [24] 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 my 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, I 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.

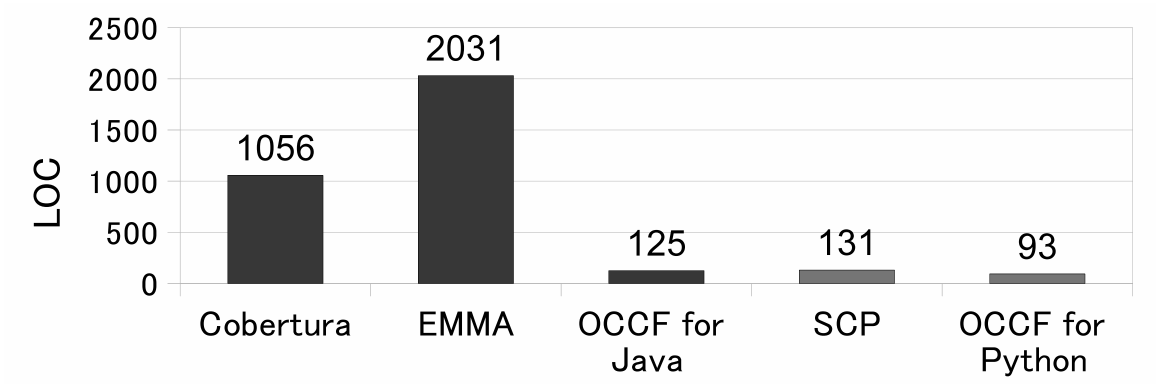


Figure 3-16: LOCs for five different tools

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

I tested five master's degree students studying computer science, who are able to read and write C and Java code. I explained my framework to them in 50 min and then provided them with the AST-generation and the AST-refinement components for C, which I implemented for them in 40 min. Table 3.3 lists the number of people who 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

Table 3.3: 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)	-

min. The examinees attempted to extend GCC to measure the two coverage criteria. However, nobody completed this task within 300 min. "-" in Table 3.3 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 3.1; thus, the same functionality was implemented with fewer LOCs.

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

3.6.2 Reduced maintenance cost

I 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: I 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. I evaluated the cost of maintenance required to extend the existing decision coverage.

I tested five master's degree students as those in experiment 3. I 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 3.3 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: I 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, I 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. I evaluated the cost of developing a new coverage criterion and the cost of the maintenance required to change the language specifications.

I tested four first year master's degree students studying computer science, who were not the examinees in experiment 4. I explained my framework to them in 30 min and then provided them with the AST-generation component for Python 3, which I implemented for them in 25 min. Table 3.3 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 I 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, I 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.

3.6.3 Consistency in measurement

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

I 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++ [63]. 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, I solved the problem (P_3) with the inconsistency in measurement.

3.6.4 Flexibility in measurement

I 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, I confirmed that developers can add new coverage criteria in the experiment 4 and 5.

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

3.6.5 Completeness in measurement

I 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 Figure 3-2 in Section 3.3.

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 Section 3.3.3 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, I have solved the problem (P_5) of incompleteness in measurement.

3.6.6 Time efficiency

I evaluated the time efficiency by using the time to execute three Java programs presented in a book [34]. This evaluation provided good results compared with the existing tools although I are not referring to the problem corresponding to this evaluation.

Experiment 6: I measured the time to execute three Java programs presented in a book. Measuring the 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 a TCP/IP communication compared with the former source code, as shown in Table 3.4. 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, I confirmed that there were no problems with the decrease in execution efficiency of the test when using OCCF with a TCP/IP communication.

3.7 Limitations

Supportable programming languages: The approach of inserting instrumentation code into source code cannot be applied to several non-procedure-oriented

Table 3.4: 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

languages. When OCCF cannot insert instrumentation code into a location where the instrumentation code is executed just before executing target program elements, OCCF cannot support such programming languages. Moreover, OCCF cannot support programming languages which have no feature to save execution traces (e.g. whitespace). However, OCCF can support major programming languages because most of programming languages is not under these constraints.

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

3.8 Related work

Kiri et al. [51] and Rajan et al. [67] among others had similar ideas and I will now refer to their study results, and my 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.

3.9 Conclusion of this chapter

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

Chapter 4

Case study of OCCF: A Tool Detecting Duplicated Test Code Based On Test Coverage Supporting Multiple Programming Languages

4.1 Introduction of this chapter

Software testing (or simply testing in what follows) is used to find defects that have been mixed into software during its development. The importance of software in society has increased, and software that is free of defects is urgently required. Therefore, testing is crucial in the development process. Production code is source code that is described as released software and test code is source code that is designed to test the production code.

Testing technology is discussed in the following, which has developed along with the increased importance of testing.

Test Driven-Development (TDD) [4] is one practice in eXtream Programming

[5], which is a kind of agile method of developing software. Developers repeat five procedures in TDD: 1) add failing test code, 2) run all tests and checking whether a new will one fail, 3) write various production codes, 4) run automated tests and check whether they have succeeded, and 5) re-factor production code. Therefore, the developer can develop test code by maintaining its state.

Test code came to be written to enforce testing as such test technologies developed. However, examples where duplicated test codes were written increased. For instance, it has been reported that there was 83.3% duplication in a test code in WinMerge open source software [74].

Duplication of source code generally deteriorates the quality of software, especially decreasing its reliability and increasing maintenance [64]. Test code is not included in the packages released to users unlike production code. However, duplicated test code also causes a decrease in the quality of software because implementation and maintenance are necessary for the test code as well as the production code.

Duplicated test code also causes a decrease in the quality of software because implementation and maintenance are necessary for the test code as well as the production code. Duplicated test code represents “a smell” in refactoring, and recommendations have been made to remove duplication [18] [60]. Moreover, the principle of “Don ’ t repeat yourself” has been recommended to avoid duplication in software including documents [37].

Duplication in test code is not only based on syntax but semantics is also a problem. One duplicated test code is redundant when two test codes detect defects that are completely identical. However, it is difficult to analyze what influence removing test code will have or to detect duplicated test code based on semantics. Therefore, no techniques have yet been established except for manually reading test code. Duplicated test code based on semantics is defined in Section 4.3.2.

Duplicated test code based on semantics causes two problems because of the extra test code.

- **Decrease in execution efficiency of testing**

  The execution efficiency of testing decreases in proportion to the amount of test code. Therefore, the execution of testing become difficult because of the increased execution time. The increase in the execution time of testing, especially in TDD or refactoring, is directly connected to the decrease in efficiency of the implementation process because testing needs to be executed frequently with the description of the production code.

- **Decrease in maintenance by testing**
  The time spent in understanding the test code increases in proportion to how much of it there is. Moreover, it becomes difficult to determine the range of the test code after it has been modified. Therefore, there is a decrease in maintenance, especially that in analyzing and changing the quality of subcharacteristics.

To solve above problems, there are the minimization techniques for test case [6] [82]. Those technique select required test cases by using various metrics: test coverage, the score of mutation testing, etc. However the method of constructing the a detection tool that considers the extendability has not been established: how to expand the supported programming languages or how to expand the supported detection criteria.

I therefore propose a technique of detecting duplicated test code based on coverage supporting multiple programming languages. My technique enables duplicated test code to be detected without the influence of measured results in the coverage. I implemented a detection tool with my technique by using OCCF. I discuss an experiment on implementing a tool that mechanically detects duplicated test code using my technique. Finally, I discuss my evaluation of the benefits and limitations of my technique.

4.2 Problem with existing detection techniques

4.2.1 P_1: Detection technique based on syntax

There are existing tools that can detect duplicated code such as CCFinderX [42]. These tools detect similar or matching parts in the source code such as copy and paste, called code clones, based on syntax. These tools cannot detect duplicated test code based on semantics. However, duplicated test code based on semantics that can find similar or matching defects causes problems.

Therefore, a detection technique based on semantics is necessary to detect duplicated test code that is redundant.

4.2.2 P_2: Detection technique that can not adjust detection criteria

No technique of adjusting the detection criteria used to detect duplicated test code has yet been established. However, it is necessary to adjust the detection criteria to determine the policy for detection. For example, when developers only want to detect duplicated test code that is obviously redundant, they only have to detect it through strict detection criteria. When developers want to detect as much duplicated test code as possible, they only have to detect it with gradual detection criteria.

Therefore, it is necessary to adjust the detection criteria according to the developer policies.

4.2.3 P_3: Inconsistent detection technique by writing test code

Developers can determine whether software has been tested sufficiently by using coverage. When developers maintain software, they can use coverage as well as write test code. When removing duplicated test code influences the measured results of coverage, sufficient testing guarantees coverage when writing is lost. Consequently, it is difficult to maintain test code. However, existing techniques of detection cannot detect duplicated test code taking coverage into consideration.

Therefore, it is necessary to detect duplicated test code consistently while writing test code so that the measured results of coverage will not be influenced.

4.2.4 P_4: Support only of one programming language

Existing detection tools support only one programming language. They does not have the extendability to add the support of programming languages. To support a new language, enormous development cost is required. Therefore, it is difficult to use these tools in the project where developers use multiple programming languages.

Therefore, it is necessary to support multiple programming language and have the extendability to add the support of programming languages.

4.2.5 P_5: High computational complexity

Some existing detection techniques require high computational complexity. For example, the techniques that use the score of mutation testing have very high computational complexity because mutation testing performs testing whenever a mutation is added. Such techniques rather decrease the execution efficiency of testing because they take too much time to detect duplicated test code.

Therefore, it is necessary to have low computational complexity.

4.3 Detection of duplicated test code based on coverage

I propose a technique of detecting duplicated test code based on coverage.

I define the inclusion relation of test code by the part that the test code covers based on coverage. My technique calculates test code that is included based on its definition and it mechanically detects duplicated test code. Moreover, my technique detects duplicated test code taking into consideration the semantics of the test code by basing it on coverage. Therefore, I alleviate or solve problem P_1.

My technique detects duplicated test code that does not influence the measured results of coverage. Therefore, I alleviate or solve problem P_3. Developers can adjust the detection criteria by changing coverage according to developer policies. Therefore, I alleviate or solve problem P_2. My technique can be applied to any language as long as coverage can be measured. Moreover, I implement the detection tool by using OCCF which measures various coverage supporting multiple programming languages. Therefore, I alleviate or solve problem P_4. My technique does the easy calculation based on the set theory for the measurement result of coverage to detect duplicated test code. Therefore, I alleviate or solve problem P_5.

4.3.1 Kind of coverage

I explain coverage that can be used to detect duplicated test code in this section.

- **Statement coverage**(C_0)

  C_0 is the ratio of statements that have been executed at least once from all statements. The measurement element in C_0 is a statement.

- **Decision coverage**(C_1)

  C_1 is the ratio of conditional branching in which all branches have been executed at least once from all conditional branching. The measurement element in C_1 is a conditional branching.

- **Condition coverage**(C_2)

  C_2 is the ratio of conditional branching in which all logical terms have been evaluated both as values of true and false from all conditional branching. The measurement element in C_2 is a logical term that composes a conditional branching.

- **Condition/Decision coverage**($\mathsf{C}_{1\&2}$)

  $\mathsf{C}_{1\&2}$ is the ratio of the conditional branching in which all branches have been executed at least once and in which all logical terms have been evaluated both as values of true and false from all the conditional branching. The measurement element in $\mathsf{C}_{1\&2}$ is a conditional branching and a logical term that composes a conditional branching.

- **Path coverage**(C_∞)

C_∞ is the ratio of the execution paths that have been executed at least once from all execution paths. The measurement element in C_∞ is an execution path.

4.3.2 Definition of duplicated test code

I define duplicated test code in this section.

Definition 1: When all defects that can be found by a certain test code can be found by another test code, the defect that can be found in testing does not change even if the test code is removed. Let us call this test code a duplicated test code based on semantics.

It is difficult to prove test code can find all defects because it is possible for innumerable software defects to exist. This causes problems with how much test code is necessary. Therefore, coverage is used as an index to determine whether testing has been completed. It is also similarly difficult to prove whether defects found by several test codes are equal.

Definition 2: Focusing on only the measurement elements of coverage, let us call the part of the production code that is executed by a certain test code in testing the part that the test code covers. When the part that a certain test code covers is covered by another test code, removing the test code does not change the measured results of coverage. Therefore, defects that can be found in testing guaranteed by the measured results of coverage do not change. Let us call this test code a duplicated test code based on coverage.

When the parts that two test codes cover are different, no defects in the part not mutually covered can generally be found. However, when the parts that two test codes cover are equal, it is not possible to discover defects equally for elements other than the measurement element of coverage even though it is possible to find defects equally for the measurement element of coverage. Therefore, the detection of duplicated test code based on coverage is the approximate detection of duplicated test code based on semantics.

Developers can only detect duplicated test code near duplicated test code based on semantics by basing it on strict coverage. For example, duplicated test code based on C_∞ includes duplicated test code based on C_0. This inclusion originates in the inclusion relation of the measurement element of coverage so that test code that covers production code by 100% based on C_∞ consistently covers production code by 100% based on C_0. There is an inclusion relation between the set of defects that can be found by duplicated test code in Figure 4-1.

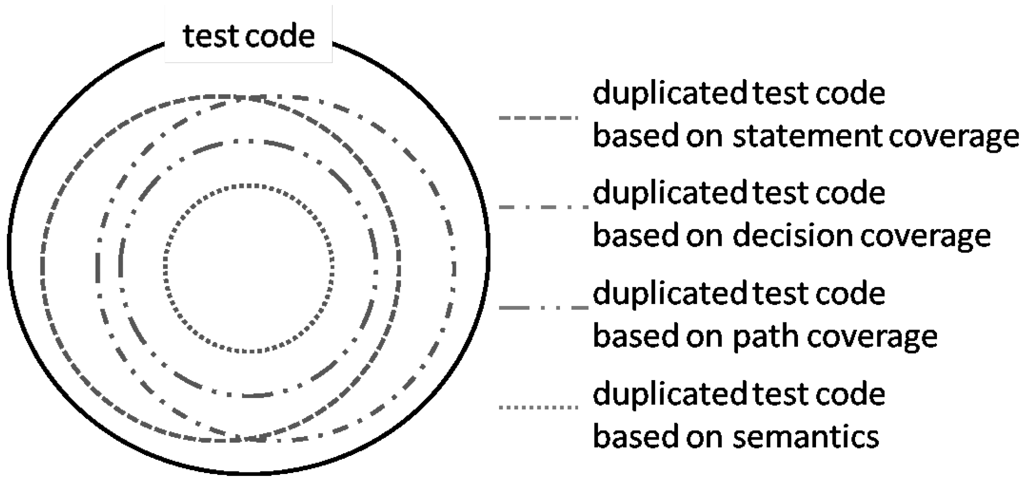

test code
duplicated test code based on statement coverage
duplicated test code based on decision coverage
duplicated test code based on path coverage
duplicated test code based on semantics

Figure 4-1: Inclusion relation between set of defects that can be found by duplicated test code

```java
String fizzBuzz(int x) {
  String str = x + ":";
  if (x % 3 == 0) {
    str += "Fizz";
  }
  if (x % 5 == 0) {
    str += "Buzz";
  }
  return str;
}
```

Figure 4-2: Production code of FizzBuzz problem written in Java

For example, there are production and test codes for the FizzBuzz problem written in the Java and JUnit4 programming languages of the testing framework in Figures 4-2 and 4-3. The FizzBuzz problem requires a program that outputs ”Fizz” when the input is a multiple of 3, it outputs ”Buzz” when the input is a multiple of 5, it outputs ”FizzBuzz” when the input is a multiple of 15, and it outputs nothing when there is no correspondence to any case. The method that has @Test annotation is

```java
@Test public void T1_Input1_Output1() {
  assertThat(fizzBuzz(1), is("1:"));
}
@Test public void T2_Input2_Output2() {
  assertThat(fizzBuzz(2), is("2:"));
}
@Test public void T3_Input3_Output3Fizz() {
  assertThat(fizzBuzz(3), is("3:Fizz"));
}
@Test public void T4_Input5_Output5Buzz() {
  assertThat(fizzBuzz(5), is("5:Buzz"));
}
@Test public void T5_Input15_Output15FizzBuzz() {
  assertThat(fizzBuzz(15), is("15:FizzBuzz"));
}
@Test public void T6_Input1and15_IsNoProblem() {
  assertThat(fizzBuzz(1), is("1:"));
  assertThat(fizzBuzz(15), is("15:FizzBuzz"));
}
```

Figure 4-3: Test code of FizzBuzz problem written in JUnit4

test code and I call the test code the head two characters of the method name in the following so that I call the test code of `T1_Input1_Output1` method `T1`.

The technique based on syntax cannot detect duplicated test code in the test code as shown in Figure 4-3. However, both `T1` and `T2` cover the statements of the 2nd and 9th lines. The measured results of C_0 do not change even if `T2` is removed Therefore, `T2` is duplicated test code.

4.3.3 Inclusion relation of test code based on coverage

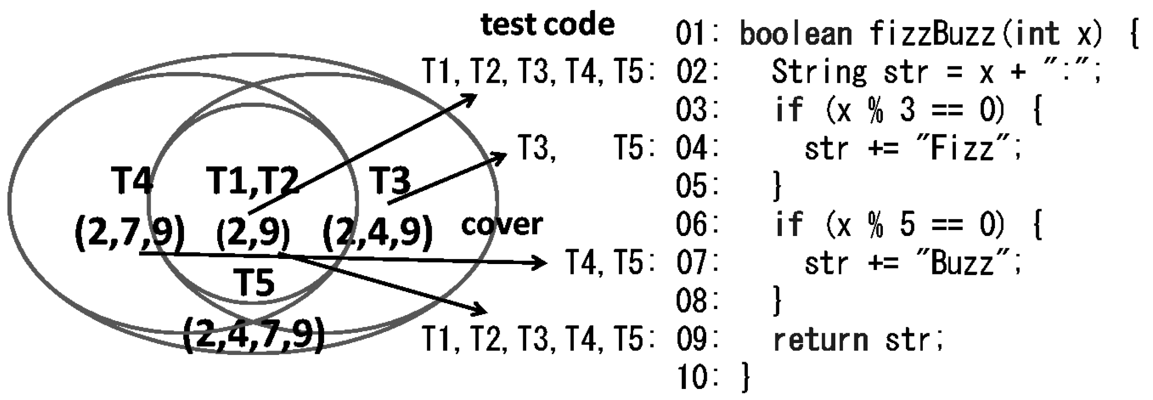
I define the inclusion relation of test code based on coverage in this section.

Definition 3: Let us call the set of measurement elements in production code covered based on coverage C when certain test code A is executed $E_C(A)$. Moreover, the inclusion relation between test code is defined equal to the inclusion relation between set $E_C(A)$. For example, for the inclusion relation between test codes A and B are based on C_0, if $E_{C_0}(A) \subset E_{C_0}(B)$, test code B includes A.

Definition4: I define the inclusion relation for the relation of the combination of two or more test code. When combination A of the test code is composed of test codes A_1, A_2, ..., and A_n, let $EC_C(A) = E_C(A_1) \cup E_C(A_2) \cup \ldots \cup E_C(A_n)$. Moreover, the inclusion relation between the combination of test code is defined as being equal to the inclusion relation between sets $E_C(A)$.

The number of measurement elements that one test code has based on coverage other than what C_∞ covers is 0 or more. However, the number of measurement elements that one test code based on C_∞ covers is always one because the measurement element based on C_∞ is an execution path.

Definition 5: I define C'_∞ that is more gradual than the detection criterion of C_∞. When the execution path of a certain test code is the partial row of the execution path of another test code, the two test codes are not equal based on C_∞. However, the definition of C'_∞ means that the partial row is already covered and the two test codes are equal based on C'_∞. I only adopt C'_∞ as the detection criterion.



```
01: boolean fizzBuzz(int x) {
02:   String str = x + ":";
03:   if (x % 3 == 0) {
04:     str += "Fizz";
05:   }
06:   if (x % 5 == 0) {
07:     str += "Buzz";
08:   }
09:   return str;
10: }
```

Figure 4-4: Inclusion of test code

For example, there are production code and test code of FizzBuzz problem in Figures 4-2 and 4-3.

Based on C_0, both `T1` and `T2` cover the statements of the 2nd and 9th lines and `T3` covers the statements of the of 2nd, 4th and 9th lines. Thus, `T3` includes `T1` and `T2`. Moreover, `T4` covers the statements of the of 2nd, 7th and 9th lines and `T5` covers the statements of the of 2nd, 4th, 7th and 9th lines. Thus, the combination of `T3` and `T4` includes `T5`. Therefore, the measured results of C_0 do not change even if `T1`, `T2`, and `T5` are removed and `T3` and `T4` are left. There is an image related to the inclusion of test code based on C_0 in Figure 4-4.

Based on C_∞, `T1` covers the path of $2 \rightarrow 9$ and `T6` covers the path of $2 \rightarrow 9 \rightarrow$

$2 \to 4 \to 7 \to 9$. Therefore, neither `T1` nor `T6` are in the inclusion relation.

The part where test code in the inclusion relation covers production code based on coverage is duplicated and the test code is duplicated test code.

4.3.4 Implementation of tool for detecting duplicated test code

I implemented a tool for detecting duplicated test code by using OCCF. The detection tool detects test code that is in the inclusion relation as duplicated test code by obtaining sets of covered measurement elements based on coverage when the coverage of the test code is measured.

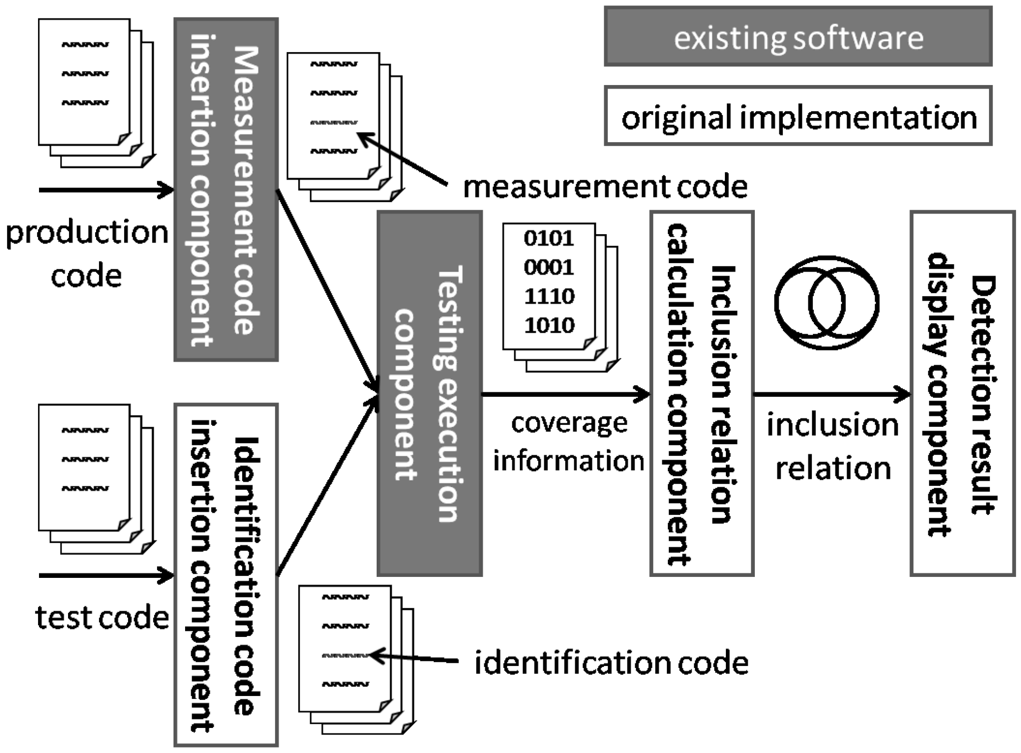

existing software
original implementation
production code
Measurement code insertion component
measurement code
test code
Identification code insertion component
identification code
Testing execution component
0101
0001
1110
1010
coverage information
Inclusion relation calculation component
inclusion relation
Detection result display component

Figure 4-5: Composition of detection tool

The composition of the detection tool is outlined in Figure 4-5. It consists of five components: the measurement-code-insertion, the identification-code-insertion, the testing-execution, the inclusion-relation-calculation, and the detection-result-display components. The measurement-code-insertion component uses OCCF and the testing-execution-component is constructed with an existing testing framework. The remaining three components are original implementations.

The detection procedure for duplicated test code with the detection tool involves five steps.

1. **Insertion of coverage-measurement code**

   The detection tool inserts special code to measure coverage (called measurement code in what follows) in the production code by using OCCF.

2. **Insertion of identification code for test code**

   The detection tool inserts special code to identify the test code to which the covered measurement element corresponds (called identification code in what follows) into the test code.

3. **Obtaining sets of measurement elements that test code covers**

   The detection tool obtains sets of measurement elements that each test code covers based on arbitrary coverage through the execution of testing.

4. **Calculation of inclusion relation of test code**

   The detection tool calculates the inclusion relation of the test code by using the sets that are obtained.

5. **Detection of duplicated test code**

   The detection tool shows the test code included to another test code as the detection results.

OCCF measures coverage by inserting the measurement code into the production code and executing the production code in testing.

For example, OCCF inserts the measurement code in the production as shown in Figure 4-6. OCCF inserts the `statement_coverage` method just before each statement and memorizes the identifier uniquely related to each statement. OCCF also inserts the `decision_coverage` method into each conditional branching and each logical term and memorizes the identifier uniquely related to each measurement

```
String fizzBuzz(int x) {
  String str = x + ":";
  if (decision_coverage(3, x % 3 == 0)) {
    statement_coverage(0); str += "Fizz";
  }
  if (decision_coverage(4, x % 5 == 0)) {
    statement_coverage(1); str += "Buzz";
  }
  statement_coverage(2); return str;
}
```

Figure 4-6: Production code of FizzBuzz problem where measurement code was inserted

element. The `decision_coverage` method obtains the original expression and returns the evaluated value of the argument so that the behavior of the production code is not changed. Thus, OCCF assesses whether each measurement element was covered by a memorized identifier to measure coverage. OCCF can measure four kinds of coverage of $\mathtt{C}_0$, $\mathtt{C}_1$, $\mathtt{C}_2$, and $\mathtt{C}_{1\&2}$ by default. OCCF can obtain an execution pass, which is the measurement element of $\mathtt{C}_\infty$, by using the measurement code inserted for measuring $\mathtt{C}_0$ even though OCCF can not measure $\mathtt{C}_\infty$. Therefore, the detection tool can calculate the inclusion relation of the test code based on five kinds of coverage of $\mathtt{C}_0$, $\mathtt{C}_1$, $\mathtt{C}_2$, $\mathtt{C}_{1\&2}$ and $\mathtt{C}_\infty$ by using OCCF.

OCCF inserts the measurement code in the production code which write various programming languages through Abstract Syntax Tree (AST). OCCF can measure coverage as long as the measurement code can be inserted in the source code. Moreover, OCCF provides common code to support multiple programming languages because AST of any language almost has the same structure. OCCF currently supports three programming languages: Java, C and Python. Therefore, the detection tool supports multiple programming by using OCCF.

The detection tool also inserts the identification code in the test code to identify the measurement element that each test code covered.

For example, the detection tool inserts the identification code in the test code as shown in List 4-7. The `start_coverage` and `end_coverage` methods memorize the identifier uniquely related to each test code. The detection tool recognizes the test code enclosed by the `start_coverage` and `end_coverage` methods as one

```
@Test public void T1_Input1_Output1() {
  start_coverage(0);
  assertThat(fizzBuzz(1), is("1:"));
  end_coverage(0);
}
@Test public void T2_Input2_Output2() {
  start_coverage(1);
  assertThat(fizzBuzz(2), is("2:"));
  end_coverage(1);
}
```

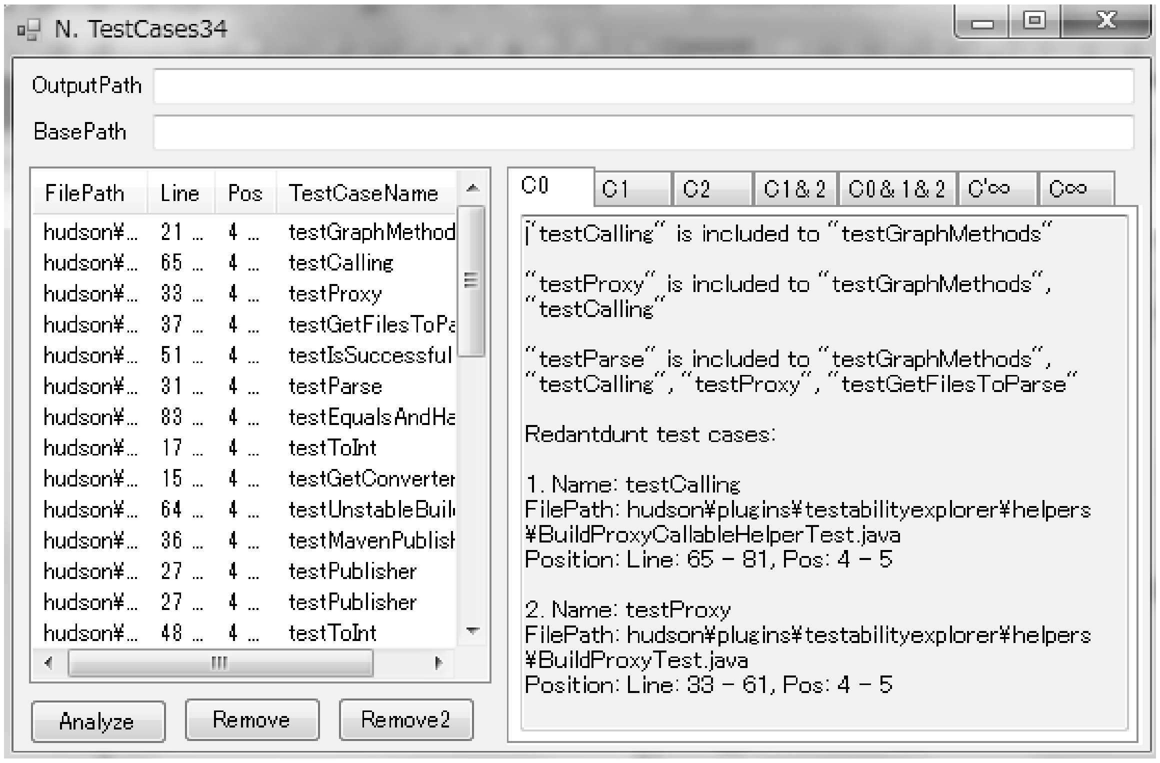
Figure 4-7: Test code of FizzBuzz problem where identification code was inserted

test code. The detection tool calculates the inclusion relation of the test code based on coverage according to set theory. The detection tool detects duplicated test code by giving a combination of test codes as one test code taking into consideration the combination. The detection tool assumes the test method described by using JUnit to be one test code and automatically inserts the identification code. The detection tool can flexibly adjust units of test code by changing how the identification code is inserted.

The inclusion-relation-calculation component obtains the measurement result of coverage. Then, this component calculates the inclusion relation of test code according to whether the measurement result of coverage is a set of measurement elements or a path of measurement elements. Therefore, this component does not depend on the kind of languages and coverage. And also, The calculation of this component has low computational complexity and the overhead of identification code is small.

There is the screen shot of the detection tool in Figure 4-8. There is a list of the test code at the left and a list of the duplicate test code at the right. The detection tool helps to remove redundant test code by revealing duplicated test code. When the test code is removed from the list at the left, the list of duplicated test code at the right is updated. Thus, developers can simulate removing test code.

Developers can remove duplicated test code by using this tool with two methods. First method removes the duplicated test code when duplicated test code is included by the other one test code. On the other hand, second method removes the duplicated test code when duplicated test code is included by the combination of the other test code. My technique greedily removes the duplicated test code. This means that my



FilePath	Line	Pos	TestCaseName
hudson¥...	21 ...	4 ...	testGraphMethod
hudson¥...	65 ...	4 ...	testCalling
hudson¥...	33 ...	4 ...	testProxy
hudson¥...	37 ...	4 ...	testGetFilesToPa
hudson¥...	51 ...	4 ...	testIsSuccessful
hudson¥...	31 ...	4 ...	testParse
hudson¥...	83 ...	4 ...	testEqualsAndHa
hudson¥...	17 ...	4 ...	testToInt
hudson¥...	15 ...	4 ...	testGetConverter
hudson¥...	64 ...	4 ...	testUnstableBuil
hudson¥...	36 ...	4 ...	testMavenPublish
hudson¥...	27 ...	4 ...	testPublisher
hudson¥...	27 ...	4 ...	testPublisher
hudson¥...	48 ...	4 ...	testToInt

```
"testCalling" is included to "testGraphMethods"

"testProxy" is included to "testGraphMethods",
"testCalling"

"testParse" is included to "testGraphMethods",
"testCalling", "testProxy", "testGetFilesToParse"

Redantdunt test cases:

1. Name: testCalling
FilePath: hudson¥plugins¥testabilityexplorer¥helpers
¥BuildProxyCallableHelperTest.java
Position: Line: 65 - 81, Pos: 4 - 5

2. Name: testProxy
FilePath: hudson¥plugins¥testabilityexplorer¥helpers
¥BuildProxyTest.java
Position: Line: 33 - 61, Pos: 4 - 5
```

Figure 4-8: Screen shot of detection tool for duplicated test code

technique remove the duplicated test code whenever duplicated test code that can be removed is found.

4.4 Detection experiment on duplicated test code with detection tool

I experimented on the detection of duplicated test code with the detection tool.

4.4.1 Application to illustrated production code and test code

There are production and test codes of the FizzBuzz problem in Figures 4-2 and 4-3. This program implemented with TDD and test code has redundant duplication, e.g., `T1` and `T2` test code for numbers that are neither multiples of three nor multiples of five. Thus, the execution paths for these test codes are equal. The development

Table 4.1: Inclusion relation of test code for FizzBuzz problem

	T1	T2	T3	T4	T5	T6
T1		C_0,C_1,C_∞	C_0	C_0	C_0	C_0,C_1
T2	C_0,C_1,C_∞		C_0	C_0	C_0	C_0,C_1
T3					C_0	C_0,C_1
T4					C_0	C_0,C_1
T5						C_0,C_1
T6					C_0	

procedure for TDD is to minimize the repeated writing of test code in which a test has failed and minimize the writing of production code that has passes the test. Thus, it is easy for the duplicated test code to be written in TDD. It is not only TDD that has a problem with duplicated test code. This problem is caused by various factors like the case where test code written in integration testing includes test code written in unit testing.

The inclusion relations of all test codes are listed in Table 4.1. The items at the far left of the table are the test codes of targets for the detection of duplicates and the items at the very top of the table are test codes where duplicated test code is included. C_0, C_1 and C_∞ in the table indicate that duplication was detected based on C_0, C_1, and C_∞.

For example, `T3` is duplicated with `T5` and `T6` based on C_0 and `T3` is duplicated with `T6` based on C_1. The measurement result of C_0 and C_1 does not change even if test code other than `T6` is removed. Moreover, `T1` and `T2` are duplicated mutually based on C_∞. The measurement result of C_∞ does not change even if either of `T1` or `T2` is removed.

The detection results for C_0 and C_1 contain all the detection results for C_∞. Thus, the detection results can be limited by basing them on the coverage of a stricter criterion. For example, when developers write test code based on 100% coverage of C_0, the measurement results do not change even if the duplicated test code that is detected based on C_0 is removed. However, developers only have to detect duplicated test code based on C_∞ to leave the test code from the viewpoint of C_∞. The developer

Table 4.2: Detection experiment on duplicated test code in open source software

	N. test code	C_0	C_1	C'_∞	C_∞	C'_0	C'_1
exp	34	3	4	1	0	32	33
task	42	8	3	2	2	27	25
tfs	190	7	8	6	5	179	183

Table 4.3: The score of mutation testing (N. detected mutations, org is original test code)

	N. mutations	org	C_0	C_1	C'_∞	C_∞	C'_0	C'_1
exp	389	150	139	148	140	148	3	3
task	113	46	44	46	46	46	25	29
tfs	444	283	281	280	283	283	51	202

only has to determine the coverage of the detection criterion according to the policy taking into consideration the trade-off in the problem that an increase in the test code causes and the sufficiency of testing.

4.4.2 Application to open source software

The results for the detection experiment on duplicated test code in open source software with the detection tool are listed in Table 4.2. And also, The results for mutation testing on minimized test code in open source software with the detection tool are listed in Table 4.3. The items at the far left of the table are the open source software of the targets for detecting duplicates. The open source software used in this experiment contained all Hudson plug-ins and was a continuous integration tool. The `exp` indicates Testability Explorer Plug-in [45], `task` indicates Task Scanner Plug-in [33] and `tfs` indicates Hudson Team Foundation Server Plug-in [76]. The items at the very top of the table indicate the number of test codes and the number of times duplicated test code was detected based on C_0, C_1, C'_∞, and C_∞. And also C_0 and C_1 indicate the tool removes the duplicated test code when duplicated test code is included by the other one test code. The C'_0 and C'_1 indicate the tool removes the duplicated test code when duplicated test code is included by the combination of the other test code.

There is little duplicated test code based on C_∞ in any of the software. However, duplicated test code based on C_0 is detected even in software with a small amount of test code. When developers have a policy of only covering 100% of C_0, developers can detect and remove a great deal of duplicated test code.

Basically, the more the number of test code decreases, the more the result of mutation testing worsens. However, the result of mutation testing of C_1 is better than C_0 though the number of test code of C_1 is lesser than C_0. The result of mutation testing of C_1 is almost the same C'_∞ and C_∞ because decision coverage is more like path coverage than statement coverage. C'_0 and C'_1 detect duplicated test code too much. Developers can select the detection criteria in consideration of this result.

4.5 Evaluation

I evaluated my technique by referring to the problems in Section 4.2.

4.5.1 S_1: Detection technique based on semantics

My technique obtained parts where the test code covered the production code based on coverage, calculated the inclusion relation of test code by using the covered parts that were obtained, and detected duplicated test code. My technique used dynamic information obtained by executing testing as well as the measurement of coverage. Thus, it could detect duplicated test code based on dynamic aspects of test code, i.e., semantics.

Therefore, I solved the problem with the detection technique based on syntax pointed out in P_1.

4.5.2 S_2: Detection technique that can adjust detection criteria

I demonstrated that my technique could adjust detection criteria by changing the kind of coverage used in detecting duplicated test code taking into consideration the trade-off in the problem that an increase in the test code causes and the sufficiency of testing. Moreover, I revealed the relation between the kind of coverage and detected duplicated test code with the implemented detection tool and my technique could limit the detection results by adjusting detection criteria.

Therefore, I solved the problem with the detection technique that could not adjust detection criteria pointed out in P_2.

4.5.3 S_3: Consistent detection technique with writing test code

I demonstrated that my technique could detect duplicated test code that did not influence the measured results of coverage by using the same coverage that was used in writing the test code in the detection criteria.

Therefore, I solved the problem with inconsistent detection with the writing test code pointed out in P_3.

4.5.4 S_4: Support of multiple programming languages

I implemented the detection tool by using OCCF. OCCF measures coverage supporting multiple programming languages, so the detection tool also supports multiple programming languages. My technique and the detection tool can use in the project where developers use multiple programming languages.

Therefore, I solved the problem with support only of one programming language with the writing test code pointed out in P_4.

4.5.5 S_5: Low computational complexity

My technique has low computational complexity because the calculation with the set theory is simple and can do it in parallel with the measurement of coverage. Actually, the time of measuring coverage and performing testing is about a minute, on the other hand, the time of calculating the inclusion and detecting duplicated test code is about a second in the experiment.

Therefore, I solved the problem with high computational complexity with the writing test code pointed out in P_5.

4.6 Limitations

I will now explain two limitations with my technique.

- **Difficulty of application to black box testing**
  My technique can only detect duplicated test code from the viewpoint of white-box testing because it detects duplicated test code based on coverage. Even if test code is duplicated from the viewpoint of white-box testing, the defects that the test code can detect might differ because black-box testing does not focus on production code and developers write test code based on the specifications used in black-box testing. This problem is essentially equal to the problem where a possibility of leakage is caused in testing even if the measured result of coverage is 100%. However, my technique is useful for limiting the candidates of duplicated test code because it is possible to manually evaluate duplication from the viewpoint of black-box testing.

- **Non-corresponding duplication of test code and redundancy**
  The duplicated test code that my technique detected was not always redundant because even if the part that the test code covered was duplicated, the defects that could be found in the testing might have differed. This problem is essentially equal to that where the measured results of coverage do not guarantee sufficient testing. However, my technique is useful in limiting the number of candidates in redundant duplicated test code.

4.7 Related work

Kamiya et al. [43] proposed efficient technique of detecting code clones and implemented a tool to detect them, called CCFinder. They kept on developing it and now have a detection tool called CCFinderX. Support for multiple programming languages, support for interactive analysis, and improved performance were added to CCFinderX. Theirs is similar to my technique from the viewpoint that it solves the problem with duplicated code. Their technique detects code clones based on syntax. My technique, on the other hand, detects duplicated test code based on coverage that has a dynamic aspect. Therefore, my technique has benefits from the viewpoint of detection of duplicated test code.

Deursen et al. [18] cataloged the refactoring of test code and proposed duplicated test code as a smell that became the motivation for refactoring. They especially discussed not only duplication based on syntax such as code clones but also duplication, which means the part of the production code that is covered by test code is equal. Moreover, they recommended changing the part of the production code that was covered by test code by refactoring the test code. My technique similarly focuses on the part of the production code that is covered by test code because my technique detects duplicated test code based on coverage. Therefore, my technique supplements refactoring from the viewpoint of automatically detecting the smell of duplicated test code.

Black et al. [6] propose the technique that detects by using all-uses coverage and calculates the set of the best minimum test case. Moreover, they show the number of defects actually found at each set of the obtained test case in the experiment. Currently, my detection tool does not support data flow coverage such as all-uses coverage. However, my technique can support data flow coverage as well as C_∞. Therefore, my technique supplements their technique from the viewpoint that my technique can use their technique.

4.8 Conclusion of this chapter

I defined the inclusion relation of test code based on coverage and proposed a technique of detecting duplicated test code. Moreover, I implemented a detection tool that mechanically detected duplicated test code and did an experiment on applying it to an illustrated program and open source software. I demonstrated my technique could detect duplicated test code based on coverage taking into consideration the semantics of test code. In addition, I showed that my technique was consistent in writing test code by detecting duplicated test code that did not influence the measured results and I established my technique could change freely the detection criteria by changing the coverage that was used by detection. Moreover, the detection tool has low computational complexity and supports three languages: Java, C and Python.

Chapter 5

UNICOEN: A Unified Framework for Code Reengineering Supporting Multiple Programming Languages

5.1 Introduction of this chapter

Many programming languages are developed and became diversified. For example, C language is a historic and widespread language which was developed in 1972. In contrast, new programming languages are being developed such as Kotlin and Xtend.

There are many source-code processing tools as well as programming languages. The tools are roughly classified into two types: source code analyzers and source code transformers. For example, software metrics measurement tools and static code analyzers belong to source code analyzers. Source-code formatters and aspect-oriented programming language processors belong to source code transformers. These tools have became popular because they have important features to improve software quality with low costs [85].

Currently, many tools are developed with respect to each programming languages. For example, FindBugs [36] and JSLint [16] are static code analyzers which support only Java and only JavaScript, respectively. Moreover, AspectJ [49] and AOJS [80] are aspect-oriented programming language processors which support only Java and only JavaScript, respectively. In this way, the many-to-many relation exists between programming languages and tools have .

Common processing can be extracted from the source code analysis and transformation. However, existing frameworks provide insufficient support for reusing common code across different programming languages. Thus, the many-to-many relation causes problems as follows. P1) Implementing tools supporting many programming languages requires enormous costs. P2) Each tool which provides same features with respect to each different programming language differs. Therefore, users of several programming languages cannot benefit from tools owing to the lack of tool support for such programming languages.

In this chapter, I propose a framework for reengineering source code supporting multiple programming languages, called UNified code reENgineering framework (UNICOEN). S1) UNICOEN provides a common model called a unified code model for representing source code in supported programming languages. S2) UNICOEN also provides a feature for inter-converting source code and objects on the unified code model. To develop source code analyzers and transformers, users can implement analysis and transformation processing with operations on the unified code model. S3) To develop tools and add extensions for supporting new programming languages, UNICOEN provides two APIs as common code on the unified code model. In this way, UNICOEN reduces development costs of tools supporting multiple programming languages to alleviate P1. Moreover, the unified code model prevents tools from differing by reusing source code processing in all programming ed by UNICOEN to alleviate P2.

We developed extensions for supporting seven programming languages and three source-code-processing tools. We compared costs for developing tools and adding extensions for new programming languages between UNICOEN and existing tools. As a results, we found that the developed tools with UNICOEN have less differences with lower costs. Thus, we confirmed that UNICOEN successfully alleviates P1 and P2.

The contributions of this chapter are as follows:

- I designed the unified code model by analyzing specifications of seven programming languages, structuring same and different features and generating a union of the language features.
- I and my developer team developed UNICOEN including two APIs for adding extensions for new programming languages and developing tools.
- I and my developer team implemented extensions for seven programming languages and developed sample tools with UNICOEN.

5.2 Problems of existing tools

5.2.1 P1: Enormous development costs

It requires enormous costs to develop tools or tool sets such that almost programming languages are supported by tools and users of almost programming languages can benefit from such tools.

Importing the tool for the other programming languages requires long time often after a tool is developed for a specific programming language. Or no tool for several programming languages is imported. Therefore, tool developers consume significant efforts to support many programming languages as far as possible. Tool users cannot apply tools to software development or they must select limited programming languages for developing software owing to the lack of the programming language support.

For example, Lint[39] which is a static analyzer for only C was released in 1977, JSLint supporting only JavaScript was released in 2002 and Pylint supporting only Python was also released in 2004. Note that JavaScript and Python were released in 1995 and 1990, respectively. However, no tool supporting Ruby exists as we investigated. This example indicates that tool users benefit from only tools corresponding to limited programming languages and that spanning of supported programming languages requires long time.

5.2.2 P2: Differences between tools

Recently, software development using multiple programming languages has been increasing. Karus et al. [44] reported developers worked with more than 4 different languages including make and XML in a project on average. Most of tools support one programming languages. Combining tools is sometimes required to apply tools to a project using multiple programming languages. However, differences between tools prevent developers from benefiting from tools or additional costs are required to combine tools. Moreover, developers use many programming languages across projects because they can select many programming languages which have different optimized areas. Developers sometimes cannot use the tool which is used in old projects in new projects owing to differences of supported programming languages between tools. Developers must consume efforts to learn how to use them in such case because developers must use the other tools in new projects. Such learning costs also prevent developers from benefiting from tools.

For example, EMMA [69] of a coverage measurement tool supports only Java and Coverage.py [2] of a coverage measurement tool supports only Python. EMMA measures statement coverage by a statement unit, while Coverage.py measures statement coverage by a line unit. It is difficult to measure consistently statement coverage owing to the unit difference between EMMA and Coverage.py when we targets software which consists of Java in server side and Python in client side with the client-sever model.

Moreover, AspectJ and AOJS of AOP processors for only Java and only JavaScript, respectively, We should write aspects for AspectJ and AOP separately when we targets software which consists of Java in server side and Python in client side similarly. This separation decreases modularity and increases learning costs of AOP processors because AspectJ and AOJS have different language specifications. Aspects for logging execution of each method is shown in Figure 5-1. The lines 1-4 and 6-9 indicate aspects in AspectJ and AOJS, respectively. Although both aspects have same features, they have different code.

```
public aspect Logger {
  pointcut all() : execution(* *.*());
  before() : all() { System.out.println(thisJoinPoint.getSignature() + " is executed."); }
}

<?xml version="1.0" ?>
<aspectsetting><function functionname="/*" pointcut="execution">
  <before><![CDATA[console .log ( __name__ + " is executed ."); ]]></before>
</function></aspectsetting>
```

Figure 5-1: Logging code for executing methods in AspectJ and AOJS

5.3 Overview of UNICOEN

Figure 5-2 shows an overview of UNICOEN. UNICOEN provides the unified code model (UCM) as a core component and both the API for adding extensions for supporting new programming languages and the API for developing tools as reusable code. UNICOEN aids users to develop tools for analyzing and transforming source code through abstract syntax trees (AST). In particular, UNICOEN aids to develop tools which mainly utilize syntax analysis instead of semantic analysis because UNICOEN structures source code on UCM by syntax analysis and a part of semantic analysis.

UNICOEN is developed in C# 4.0, thus, runs on .NET Framework and Mono. UNICOEN is a open source licensed by Apache 2.0 and is available on Github[70]

S1) UNICOEN provides UCM as a specification of a common AST for supported programming languages. Common objects on UCM for supported programming languages are called unified code objects. UNICOEN provides the API for developing tools which is similar to DOM standardized by W3C and LINQ developed by Microsoft. In particular, the API provides a feature for inter-converting between source code and objects on UCM in a same programming languages, and basic operations of a extraction, addition, replacement and deletion for analyzing and transforming code. Operations for objects on UCM aids to commonalize soure-code processing

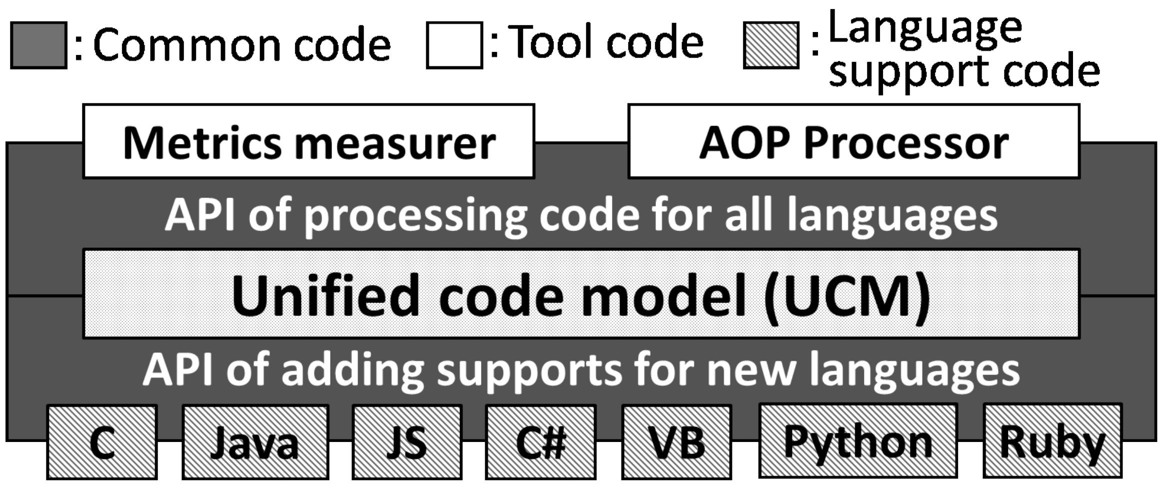


Figure 5-2: Overview of UNICOEN

for common elements between supported programming languages. Commonalizing source-code decrease differences between tools. Therefore, UNICOEN alleviates P2) the differences between tools supporting different programming languages.

UNICOEN also provides the API for adding extensions for supporting new programming language, which requires users to implement a mapper between source code an objects on UCM called a object-code mapper with respect to each programming languages. To aid to implement extensions, UNICOEN provides reusable code for implementing object-code mappers as the API. S2) UNICOEN provides the interconversion between source code and objects on UCM and S3) provides the two APIs as common processing to reduce costs of development tools and extensions for supporting new programming languages to reduce both costs of developing tools and extensions. Therefore, UNICOEN alleviates P1) enormous costs to implement tools supporting many programming languages.

UNICOEN users are classified into two types: tool developers which develop tools with the API for developing tools and extension developers which add extensions for supporting new programming languages in UNICOEN with the API for adding extensions for supporting new programming language. The tool developers can avoid implementing syntax analysis and a part of semantic analysis by implementing code processing on UCM. Moreover, the tool developers can develop any tools with the same API by using UCM and the API for developing tools. Whereas, the extension developers can extend UNICOEN to support more programming languages by implementing object-code mappers.

5.4 Implementation of UNICOEN

This section describes UCM, the API for adding extensions for supporting new programming language and the API for developing tools, respectively.

5.4.1 Unified code model

UCM is defined with classes in C#. Objects on UCM, which are instances of the classes, have recursive tree structure. For example, an object for representing a class has children objects for representing methods and the children have children objects such as a parameter and a block. The objects also have position information on source code (a line and a row number).

UCM structures source code as objects mainly based on syntax analysis and UNICOEN does not requires full semantic analysis. For example, UNICOEN recognizes a syntax of a binary expression, on the other hand, it does not interpret a meaning of a binary expression. Thus, UNICOEN is different from compiler frameworks such as GCC, LLVM and virtual machines, which executes intermediate code, such as Java VM and .NET Framework. Note that Section 5.6 describes these tools.

I designed UCM which have the capability to represent source code of C, Java, C#, Visual Basic, JavaaScript, Python and Ruby by integrating language features and grammars of these programming languages. I consider elements of these programming languages which have similar syntax and meaning as common elements on UCM. I structured UCM by calculating the union of the common elements and others.

For example, a `while` statement in most of programming languages has a condition whether a loop continues and a imperative block in the loop. However, a `while` statement in Python has an else-clause which has a imperative block which is executed when the condition is false and the loop terminates. To represent both `while` statements, UCM has a `while` statement which has the three elements: a condition, a imperative block and an else-clause. UCM considers a package declaration in Java and a namespace declaration in C# as a same element because they have similar meanings although a package declaration in Java and a namespace declaration in C# have different notation styles. Moreover, UCM considers a package declaration, a namespace declaration, a class declaration including an interface as a similar elements because some programming languages allow namespaces to contain fields and methods directly as well as a class declaration.

I judged whether elements of these programming languages are common or not from similarities of names, structures, meanings and positions. Steps to find candidates of common elements for constructing UCM from programming language A and B is described as follows.

1. Finding the most abstract non-terminal symbol in programming language A
2. Comparing non-terminal symbols which are not candidates in programming language B with the found symbol with breadth-first search (BFS) to find elements which meet the following requirements. Note that the search starts from the child symbols of the candidate when the parent symbol of the found symbol has a candidate of common elements.
    - The names of the non-terminal symbol are similar.
    - The child symbols of the non-terminal symbol are similar and the structures of the non-terminal symbol are similar.
    - The meanings and positions of the non-terminal symbol are similar.
3. The pair of non-terminal symbols are considered as a candidate of common elements when the elements which meet either of requirements are found.
4. The step 2 are repeatedly applied to the child symbols in programming language A with BFS. Terminating the steps when no child symbol is found.

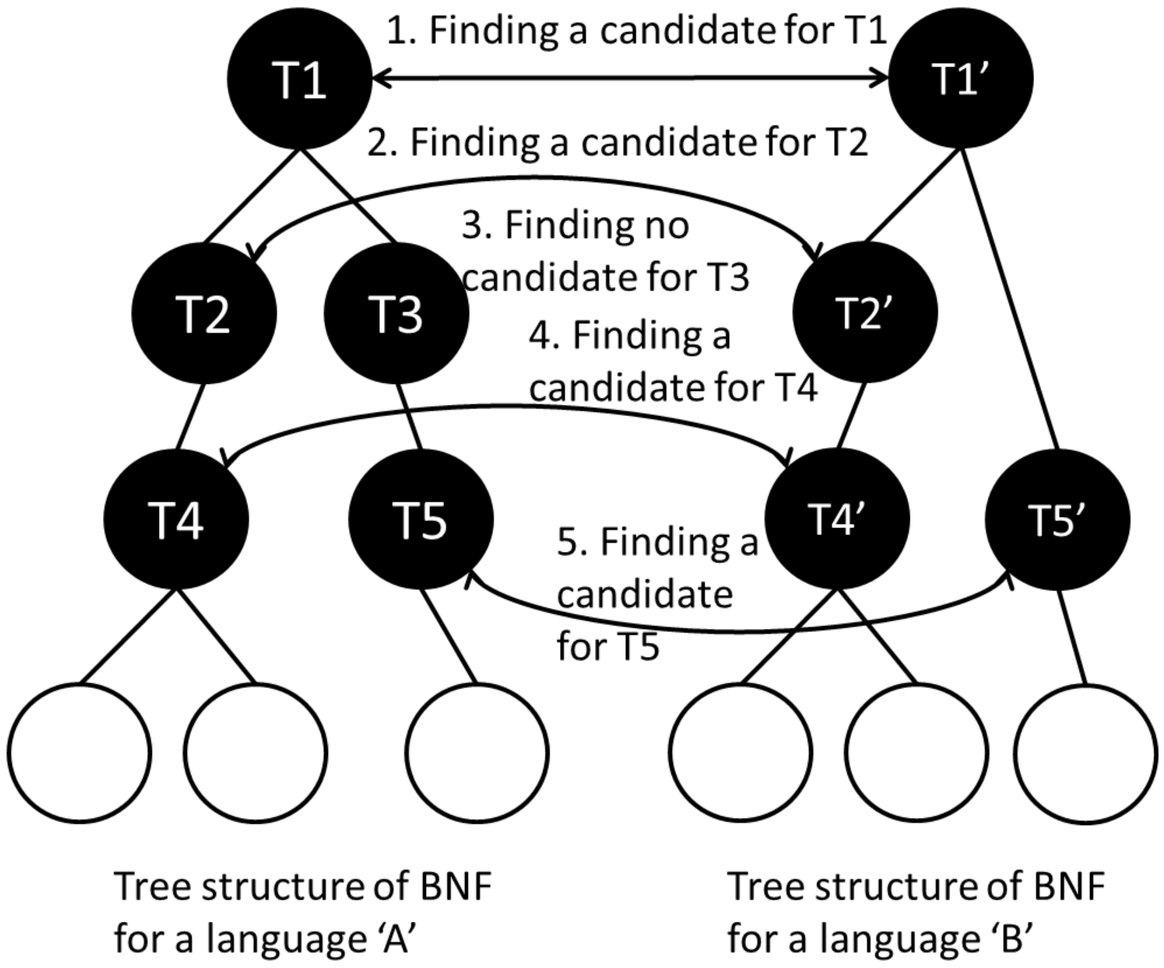

1. Finding a candidate for T1
2. Finding a candidate for T2
3. Finding no candidate for T3
4. Finding a candidate for T4
5. Finding a candidate for T5
T1
T1'
T2
T3
T2'
T4
T5
T4'
T5'
Tree structure of BNF for a language 'A'
Tree structure of BNF for a language 'B'

Figure 5-3: Illustration of selecting common elements

Figure 5-3 shows an example for extracting common elements from programming languages A and B, which have five and four non-terminal symbols, respectively. Black circles indicate non-terminal symbols and white circles indicate terminal symbols. Pairs of T1 and T1', of T2 and T2', of T4 and T4' and of T5 and T5' are candidates of common elements. First, finding a candidate of common elements for T1 starts, then, T1' is found. Second, finding a candidate of common elements for T2 starts. T2' is found first because T1 is the parent symbol of T2 and T1 is similar to T1'. Although T5' and T4' are compared with T3, a candidate of common elements for T3 is not found. Then, T4' is found first as a candidate of common elements for T4 because T2 is the parent symbol of T4 and T4 is similar to T4'. Finally, T5' is found first as a candidate of common elements for T5.

Figure 5-4 shows the partial definition of UCM written in extended Abstract Syntax Description Language (ASDL). Figures 5-12, 5-13 and 5-14 also shows the full definition of UCM written in extended ASDL. Tables 5.7, 5.8, 5.9, 5.10 and 5.11 show the relation between elements on UCM and programming languages. “x” indicates a programming languages has a element.

Most of procedural programming languages distinguish expressions and statements by judging whether they return values. However, most of elements in Ruby are expressions and Ruby has few statements which has no return values. For example, six programming languages other than Ruby have an `if` statement, a `while` statement and a function declaration, which are statements while ones in Ruby are expressions. Thus, UCM considers a statement as a special case of expressions, that is, UCM considers both statements and expressions as expressions In this way, UNICOEN represents source code as objects on the same model (UCM) and provides same operations with the API for objects on UCM which is designed to represent source code of the seven programming languages.

```
Expression = If(Expression condition, Block body, Block elseBody)
    | While(Expression condition, Block body, Block elseBody)
    | DoWhile(Expression condition, Block body)
    | For(Expression initializer, Expression condition, Expression step,
       Block body, Block elseBody)
    | FunctionDefinition(ModifierCollection modifiers, Type returnType,
       Identifier name, ParameterCollection parameters, Block body)
```

Figure 5-4: Part of unified code model in ASDL

5.4.2 API for adding extensions for supporting new programming language

UNICOEN provides two features as the API for adding extensions for supporting new programming language to reduce costs of extensions for supporting new programming languages: a feature to convert parse results from ANTLR into XML element objects on .NET Framework and a feature to scan the XML trees. The extension developers must implement object-code mappers which consists in syntax analyzers, object generators and code generators with the API to add extensions for supporting new programming language. To implement object-code mappers, the extension developers must create classes to implement interfaces provided by UNICOEN. Figures 5-5 and 5-6 show an inter-conversion between source code and objects on UCM as a usage example of object-code mappers.

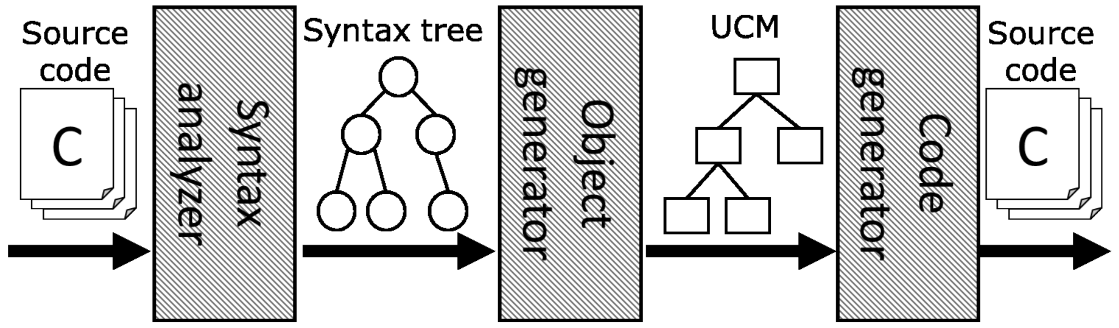


Figure 5-5: Process of conversion and reverse conversion between source code and unified code objects

```
var filePath = "code.java";
var ext = Path.GetExtension(filePath);
var progGen = UnifiedGenerators.GetProgramGeneratorByExtension(ext);
var uco = progGen.GenerateFromFile(filePath);
// Write transformation processing
var code = progGen.CodeGenerator.Generate(uco);
```

Figure 5-6: C# Code to inter-convert between source code and unified code objects with UNICOEN

UNICOEN makes it possible that objects on UCM generated from source code of supported programming languages are reconverted into source code of the same programming language without changing meanings. The extension developers must implement object-code mappers to achieve this inter-conversion. Moreover, UNICOEN provides test cases to judge whether object-code mappers achieve this inter-conversion. This inter-conversion aids to transform source code on UCM as long as the scope of the transformation does not extend beyond the expressiveness of the original programming language. UNICOEN tries to reconvert objects on UCM into source code ignoring objects which is beyond the expressiveness by default. Note that developers can implement an object-code mapper which converts a class object on UCM into source code in C which represent the class with function pointer and structure. In this way, the way of the inter-conversion is freely designed by extension developers.

UNICOEN aids the extension developers to add extensions for supporting new programming languages with ANTLR or existing parser libraries. UNICOEN also provides a sub system which aids to utilize ANTLR, called Code2Xml. Code2Xml modifies parsers generated by ANTLR to be suitable as object-code mappers. Moreover, UNICOEN provides reusable code to scan XML trees and analyze expressions as XML elements.

The extension developers can add a extension for supporting a new programming language with the following steps. 1) They investigate the specification of the programming language and design the model to structure source code. 2) They compare the designed model with UCM to find differences. 3) They extend UCM with a step a), a step b) or both steps. 3.a) When the element does not exist in UCM, they add classes for representing the element in UCM. For example, currently UCM has no element for representing aspects, thus, they can add classes for aspects. 3.b) When the element exists in UCM but some properties are not represented on UCM, they extend existing classes adding properties. For example, they can add a new property of the else-clause in the class of a `for` statement if a `for` statement has an else-clause of a `while` statement in Python.

Existing object-code mappers and tools with UNICOEN are not affected when UCM is extended. When new classes are added to UCM, the objects of the classes cannot be generated and it is impossible to judge whether the objects does not appear in the source code or the programming languages has no feature and grammar for the objects. Similarly, when new properties are added in classes of UCM, the property is initialized to null and it is also impossible to judge whether the property does not appear in the source code or the programming languages has no feature and grammar for the property.

5.4.3 API for developing tools

UNICOEN provides two features as the API for developing tools to reduce development cost of tools supporting multiple programming languages: a feature to interconvert between source code and objects on UCM and a feature to provide operations for objects on UCM such as an extraction, addition, replacement and deletion. The features are achieved by object-code mappers which the extension developers develop. Figure 5-7 shows a class diagram of the unified code model and the API for developing tools.

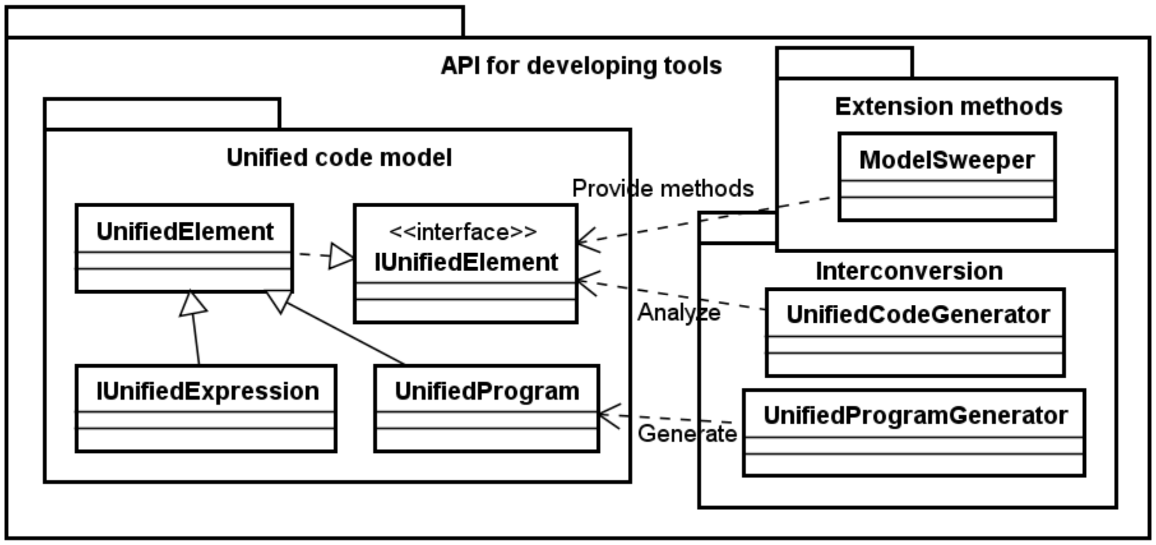


Figure 5-7: Class diagram of the unified code model and the API for tool developers

The analysis and transformation features for objects on UCM are provided by the API which is similar to LINQ to XML. Thus, the API can be combined with operations in LINQ. For example, source code which counts the XML element whose name is `if` with LINQ to XML shown in Lines 1-3 Figure 5-8 and source code which counts `if` statements with UNICOEN shown in Lines 5-7 Figure 5-8 are very similar. The types of the `ifElements` and `ifs` in Figure 5-8 are The `IEnumerable<XElement>` and the `IEnumerable<IUnifiedElement>`, respectively. The `XElement` represents an XML element and the `IUnifiedElement` represents objects on UCM. Both types provide the `Count()` extension method of LINQ because both types are the

IEnumerable.

```
var xml = XDocument.Load("code.xml");
var es = xml.Descendants("if");
Console.WriteLine("#if elements: " + es.Count());

var uco = UnifiedGenerators.GenerateProgramFromFile("code.java");
var ifs = uco.Descendants<UnifiedIf>();
Console.WriteLine("#ifs: " + ifs.Count());
```

Figure 5-8: C# code to enumerate "if" elements and enumerate unified code objects of "if" statement

To develop analysis tools with UNICOEN, the developers can write analysis processing after conversion processing from source code into objects on UCM. In contrast, to develop transformation tools with UNICOEN, the developers can write reconversion processing from objects on UCM into source code after transformation processing of objects on UCM. For example, to measure the number of statements, developers can write source code which enumerates all blocks and counts child elements of the blocks because child elements of blocks on UCM indicate statements. Figure 5-9 shows sample code to measure the number of statements.

```
var uco = UnifiedGenerators.GenerateProgramFromFile("code.java");
var count = uco.Descendants<UnifiedBlock>().Sum(e => e.Count);
Console.WriteLine("#statements: " + count);
```

Figure 5-9: C# Code to count statements with UNICOEN

In this way, the tool developers can write program to count any element through enumerations of specific objects on UCM with the API for developing tools. Moreover, UNICOEN makes it easy to add method declarations, change operations of expressions and delete any statement by providing mutable properties, the `Add` and `Remove` methods. Therefore, UNICOEN reduces development costs of tools supporting multiple programming languages.

5.5 Evaluation

This sections describes evaluations through my case study to confirm the effectiveness of UNICOEN. I and my developer team implemented OC-mappers for seven programming languages and then developed two metrics measurement tools. I confirm UNICOEN reduces development costs to alleviate P1 by comparing the number of statements which are required to develop tools between the metrics measurement tools and existing tools. I also confirm UNICOEN reduces differences between tools to alleviate P2 by checking whether the metrics measurement tools provides same features for all supported programming languages.

5.5.1 Implementation of extensions for supporting programming languages

We developed object-code mappers for C, Java, C#, Visual Basic, JavaScript, Python and Ruby. The comparison of the number of statements between implemented object-code mappers and existing programming language processors is shown in Table 5.1. The column of object-code mapper indicates the number of statements of the object generator implemented by hand. The column of existing programming language processors indicates the number of statements for the compilation processing of GCC for C, GCJ for Java, Mono MCS for C#, Rhino which is a JavaScript processor on Java VM, IronPython which is a Python processor on .NET Framework and IronRuby which is a Ruby processor on .NET Framework. Note that the number of statements for the processors other than Rhino excludes one for the common processing of frameworks and includes only the conversion processing between source code and machine code/intermediate code. However, the value for Rhino indicates the number of statements for the entire processor because Rhino does not use any framework and it is difficult to extract the compilation processing. Moreover, Table 5.1 does not include Visual Basic because no processor for Visual Basic provides source code.

The implementation uses ANTLR for C, Java and JavaScript and existing parser libraries for other languages. UNICOEN reduce code from one twenty to one fifty

Table 5.1: Comparison of the number of statements between OC mappers and existing programming language processors: GCC, GCJ, Mono, Rhino, IronPython, IronRuby

Programming language	C	Java	C#	JavaScript	Python	Ruby
Object-code mapper	727	1,003	399	626	636	501
Existing processor	14,949	12,782	36,988	38,277	15,411	14,353

comparing with existing processors by utilizing existing software and providing the API for adding extensions for supporting new programming languages. Therefore, we confirm UNICOEN alleviates P1.

5.5.2 Tool development with UNICOEN

We developed UniMetrics which measure McCabe's complexity and result values which can be visualized as a city by CodeCity[81].

There are some existing tools for measuring McCabe's complexity: Sonar[77] for Java and Saikuro[7] for Ruby. However, the measurement tools have different for foreach statements and it requires additional efforts to merge measurement results. This situation makes evaluation of entire software written in Java and Ruby with McCabe's complexity hard. In contrast, UniMetrics can measure McCabe's complexity with same criteria for all supported programming languages and show the results in one graph. Moreover, CodeCity has metrics measurement tools for Java, C++ and C#, while CodeCity cannot integrate measurement results for software written in multiple programming languages to visualize several results as one city. In contrast, UniCodeWorld can generate one measurement result, and thus, CodeCity can visualize it as one city.

The comparison of the number of statements for measuring McCabe's complexity between Saikuro and UniMetrics is shown in Table 5.2. The comparison of the number of statements for generating measurement results for CodeCity between PMCS[19] for C# and UniCodeWorld is shown in Table 5.3. Moreover, Tables 5.4 and 5.5 show the elements which are used for measuring McCabe's complexity and generating results for CodeCity by my tools. "x" indicates a programming languages has a element.

Table 5.2: Comparisons of the supported languages and the number of statements between Saikuro and UniMetrics

	Supported programming languages	Statements
Saikuro	Ruby	321
UniMetrics	C, Java, C#, Visual Basic, JavaScript, Python, Ruby	3

Table 5.3: Comparisons of the supported languages and the number of statements between PMCS and UniCodeWorld

	Supported programming languages	Statements
PMCS	C#	1478
UniCodeWorld	C, Java, C#, Visual Basic, JavaScript, Python, Ruby	203

Both Saikuro and PMCS have processing for syntax analysis and they measures metrics in analyzing syntax. On the other hand, tools with UNICOEN can measure metrics without their own syntax analysis because UNICOEN provides reusable code to convert source code into UCOs. As shown in Tables 5.4 and 5.5, the tools with UNICOEN successfully reuse language-independent code by manipulating common elements across multiple programming languages on UCM. As a result, UNICOEN reduces development costs as shown in Tables 5.2 and 5.3.

Therefore, we confirm UNICOEN alleviate P2 by decreases differences between tools for other programming languages and alleviate P1 by reducing development costs of tools for multiple programming languages.

The number of statements which I and my developer team implemented for UNICOEN including OC-mappers without libraries and automatically generated code is 14796. Figure 5-10 and 5-11 shows graphs the number of required statements to implement tools with UNICOEN similar to Saikuro and PMCS, respectively. The horizontal axis indicates the number of combinations of programming languages and tools. Thus, this value increases when adding a new supported programming language. Note that the number of required statement are calculated on the basis of Tables 5.2 and 5.3.

Traditional tool development develops tool with respect to each programming languages without reusing code across different programming languages. Thus, I assume

Table 5.4: Elements in the unified code model for measuring McCabe complexity and languages (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Programming language					
Element	L1	L2	L3	L4	L5	L6
If	x	x	x	x	x	x
For	x	x	x	x		
Foreach		x	x	x	x	x
While	x	x	x	x	x	x
DoWhile	x	x	x	x	x	
Case	x	x	x	x	x	

Table 5.5: Elements in the unified code model for generating the measurement result for CodeCity complexity and languages (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Programming language					
Element	L1	L2	L3	L4	L5	L6
NamespaceDefinition		x	x			
ClassDefinition		x	x		x	x
FunctionDefinition	x	x	x	x	x	x
VariableIdentifier	x	x	x	x	x	x
Modifier	x	x	x	x	x	x

that the required statements when adding a new supported programming language and when developing a new tool are same. I show two cases for UNICON where tools support seven programming languages and support only one programming language. An increase in required statements occurs when developing seven combinations in the case where tools support seven programming languages with UNICOEN. On the other hand, an increase in required statements occurs when developing one combination in the case where tools support only one programming languages with UNICOEN.

As shown in the graphs, developers are required to develop 47 combinations for tools similar to Saikuro to overcome required statements to develop UNICOEN. On the other hand, developers are required to develop 12 combinations for tools similar to PMCS to overcome required statements to develop UNICOEN. Therefore, developers

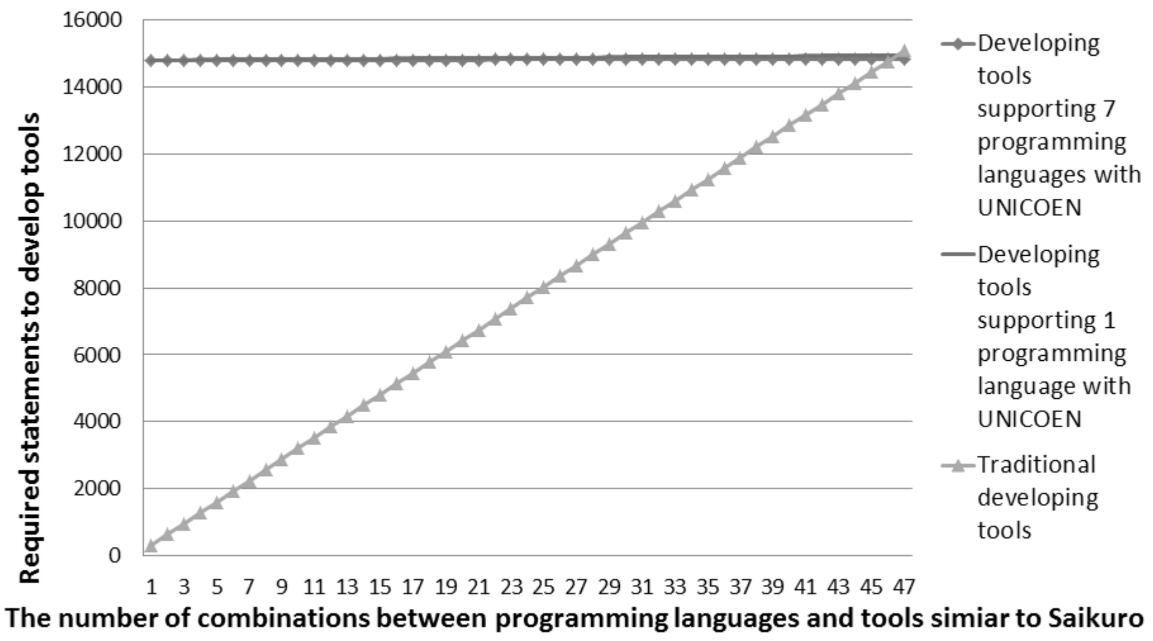


Figure 5-10: The number of statements for developing tools whose size is similar to the size of Saikuro

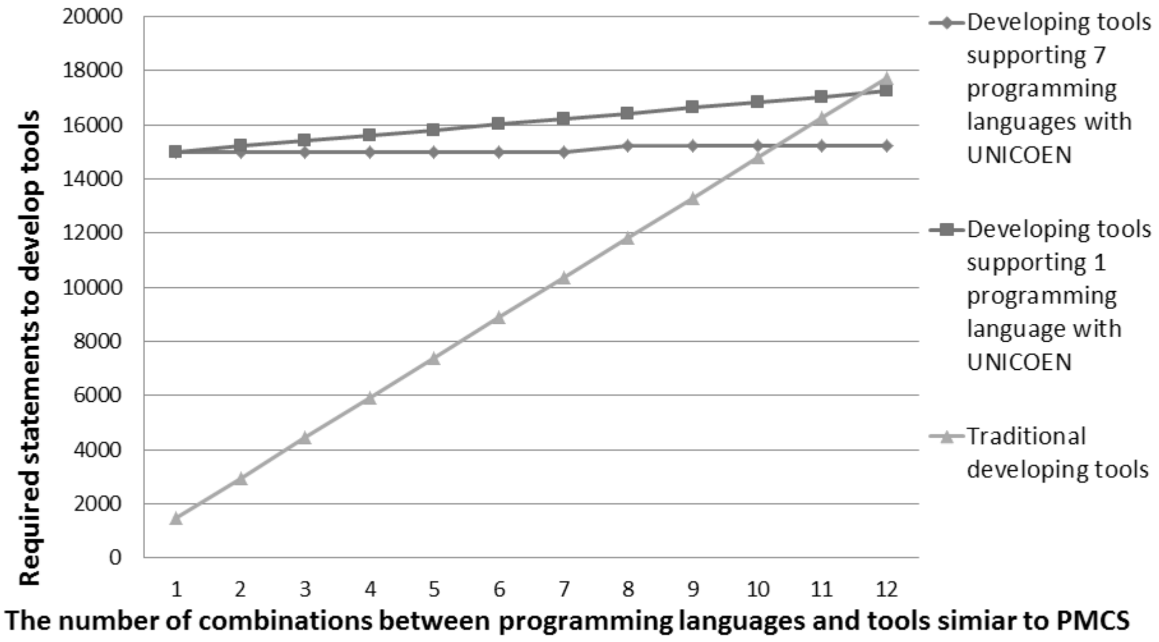


Figure 5-11: The number of statements for developing tools whose size is similar to the size of PMCS

benefit from a framework such as UNICOEN when developers develop UNICON and 50 combinations at least.

Table 5.6: Comparison of UNICOEN with existing tools and frameworks

Framework	Adding new languages	Common model	Analysis	Transformation
UNICOEN	x	x	x	x
LLVM	x	x	x	x
MASU		x	x	
Sapid			x	x
srcML		x	x	x
DMS	x		x	x
TXL	x			x
Stratego	x			x

5.6 Related works

Table 5.6 shows a comparison of UNICOEN with existing tools and frameworks. "Adding new languages" indicates whether a tool or framework supports to add extensions for supporting new programming languages. "Common model" indicates whether a tool or framework has a common language model. "Analysis" and "Transformation" indicates whether a tool or framework supports analysis and transformation, respectively.

Lattner et al. [54] proposed a compile framework, called LLVM. In addition, a similar target for comparison with UNICOEN is a platform for running intermediate languages such as Java VM and .NET Framework. Because such software analyze semantics completely and generate intermediate code, it requires enormous costs to add extensions for supporting new programming languages. Moreover, intermediate languages are similar to machine languages and contain little information about syntax. Thus, it is difficult to develop tools needing syntax information such as code formatter. UNICOEN, in contrast, reduce costs of tool development and extensions for supporting new programming languages by avoiding full semantic analysis. UNICOEN enables developers to develop tools for any supported programming languages similarly by providing UCM and the API for developing tools. UNICOEN also allows developer to implement semantic analysis for tools which requires semantic information. Therefore, UNICOEN has a significant advantage when developing tools which require syntax information mainly.

Higo et al. [35] proposed a framework for measuring metrics supporting multiple programming languages, called MASU. MASU interprets elements required to measure metrics and constructs language-independent AST. MASU allows users to implement plug-ins which analyze the language-independent AST to measure software metrics for Java, C# and Visual Basic. UNICOEN, in contrast, supports more programming languages than MASU including not object-oriented programming languages and dynamic programming languages. Moreover, whereas MASU only provides language-independent AST, UNICOEN also provides the API for adding extensions for supporting new programming languages. In addition, UNICOEN can and MASU cannot transform source code.

Baxter et al. [3] proposed a toolkit which aids to develop analysis and transformation tools, called DMS. Tool developers can develop tools with DMS and extension developers also can add new extensions for supporting new programming languages by adding grammars of new programming languages. However, DMS does not provide common language model, thus, it is hard to reuse code across different programming languages and decrease differences between tools for different programming languages.

Sapid [26] and srcML [12] are tools for analyzing and transforming source code. Although tool developers can utilize these tools for developing tools, extension developers cannot add extensions for supporting new programming languages.

TXL [15] and Stratego [8] are tools for transforming source code. Although these tools support adding extensions, developers cannot develop analysis tools with them.

5.7 Conclusion of this chapter

This chapter described UNICOEN for processing source code supporting multiple programming languages. UNICOEN aids to develop tools which utilize syntax infromation mainly by providing UCM and the two APIs for developing tools and adding extensions for supporting new programming languages. We added seven programming languages in UNICOEN: C, Java, C#, Visual Baisc, JavaScript, Python and Ruby, and developed UniMetrics and extension of CodeCity. We confirmed UNICOEN reduces costs of development tool and extensions for supporting new programming languages and UNICOEN also reduces differences between tools for different programming languages. Therefore, UNICOEN alleviate P1 and P2.

Table 5.7: Elements in the unified code model and languages (Part 1) (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Language					
Element	L1	L2	L3	L4	L5	L6
Program	x	x	x	x	x	x
Block	x	x	x	x	x	x
Comment	x	x	x	x	x	x
VariableDefinition	x	x	x	x		x
VariableDefinitionList	x	x		x		x
ClassDefinition		x	x		x	x
AnnotationDefinition		x				
EigenClassDefinition					x	
EnumDefinition	x	x	x			
InterfaceDefinition		x	x			
ModuleDefinition					x	
NamespaceDefinition		x	x			
StructDefinition	x		x			
UnionDefinition	x					
EventDefinition			x			
FunctionDefinition	x	x	x	x	x	x
PropertyDefinition			x			
PropertyDefinitionPart			x			
Constructor		x	x	x		
InstanceInitializer				x		
StaticInitializer		x	x			
TypeConstrain		x	x			
SuperConstrain		x	x			
ReferenceConstrain			x			
ImplementsConstrain		x	x			
ExtendConstrain		x	x			
EigenConstrain					x	
ConstructorConstrain			x			
AnnotationCollection		x				
ArgumentCollection	x	x	x	x	x	x
CaseCollection	x	x	x		x	
CatchCollection		x	x	x	x	x
ExpressionCollection	x	x	x	x	x	x

Table 5.8: Elements in the unified code model and languages (Part 2) (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Language					
Element	L1	L2	L3	L4	L5	L6
GenericArgumentCollection		x	x			
GenericParameterCollection		x	x			
IdentifierCollection	x	x	x	x	x	x
ModifierCollection	x	x	x	x	x	x
OrderByKeyCollection			x			
ParameterCollection	x	x	x	x	x	x
TypeCollection	x	x	x		x	x
TypeConstrainCollection	x	x	x		x	x
Modifier	x	x	x	x	x	x
Annotation		x	x			
Parameter	x	x	x	x	x	x
Argument	x	x	x	x	x	x
GenericParameter		x	x			
GenericArgument		x	x			
Call	x	x	x	x	x	x
New	x	x	x	x		
Property	x	x	x	x	x	x
Indexer	x	x	x	x	x	x
Slice						x
Identifier	x	x	x	x	x	x
VariableIdentifier	x	x	x	x	x	x
LabelIdentifier	x		x			
SuperIdentifier		x	x			x
ThisIdentifier		x	x			x
TypeIdentifier			x			
VaueIdentifier			x			
Cast	x	x	x			
Type	x	x	x	x	x	x
ArrayType	x	x	x			
ConstType	x					
GenericType		x	x			
PointerType	x					
StructType	x		x			

Table 5.9: Elements in the unified code model and languages (Part 3) (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Language					
Element	L1	L2	L3	L4	L5	L6
UnionType	x					
VolatileType	x					
NullLiteral	x	x	x	x	x	x
BooleanLiteral		x	x	x	x	x
CharLiteral	x	x	x	x		x
FractionLiteral	x	x	x		x	x
RegularExpressionLiteral				x	x	
StringLiteral	x	x	x	x	x	x
SymbolLiteral					x	
IntegerLiteral	x	x	x	x	x	x
ArrayLiteral		x	x	x	x	x
IterableLiteral						x
ListLiteral				x		x
MapLiteral				x	x	x
SetLiteral						x
TupleLiteral						x
KeyValue				x	x	x
Range					x	
BinaryExpression	x	x	x	x	x	x
TernaryExpression	x	x	x	x	x	x
UnaryExpression	x	x	x	x	x	x
BinaryOperator	x	x	x	x	x	x
UnaryOperator	x	x	x	x	x	x
Sizeof	x		x			
Typeof		x	x			
If	x	x	x	x	x	x
For	x	x	x	x		x
Foreach		x	x	x	x	x
While	x	x	x	x	x	x
DoWhile	x	x	x	x	x	
Switch	x	x	x	x	x	
Case	x	x	x	x	x	
Label	x	x		x		

Table 5.10: Elements in the unified code model and languages (Part 4) (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Language					
Element	L1	L2	L3	L4	L5	L6
Lambda			x	x		x
Proc					x	
Try		x	x	x	x	x
Catch		x	x	x	x	x
Fix			x			
Synchronized		x	x			
Using			x			
With				x		
Break	x	x	x	x	x	x
Continue	x	x	x	x	x	x
Return	x	x	x	x	x	x
Throw		x	x	x	x	x
Goto	x		x			
Redo					x	
Retry					x	
YieldBreak			x			
YieldReturn			x		x	x
Alias					x	
Assert		x		x		x
Default			x			
Defined					x	
Delete				x	x	x
Exec						x
Import		x	x			x
Pass				x		x
Print						x
PrintChevron						x
StringConversion						x
MapComprehension						x
IterableComprehension						x
ListComprehension						x
SetComprehension						x

Table 5.11: Elements in the unified code model and languages (Part 5) (L1:C, L2:Java, L3:C# and Visual Basic, L4:JavaScript, L5:Ruby, L6:Python)

	Language					
Element	L1	L2	L3	L4	L5	L6
LinqExpression			x			
LinqQuery			x			
FromQuery			x			
GroupByQuery			x			
JoinQuery			x			
LetQuery			x			
OrderByQuery			x			
SelectQuery			x			
WhereQuery			x			
OrderByKey			x			

```
IElement
= Element
| ElementCollection

Element
= Expression()
| Program(Block body)
| Parameter(AnnotationCollection annotations, ModifierCollection modifiers,
  Type type, IdentifierCollection names, IExpression defaultValue,
  IExpression annotationExpression)
| Modifier(string name)
| GenericParameter(Type type, TypeConstrainCollection constrains,
  ModifierCollection modifiers)
| GenericArgument(IExpression type, ModifierCollection modifiers,
  TypeConstrainCollection constrains)
| Comment(string comment)
| Case(IExpression condtion, Block body)
| Argument(IExpression value, Identifier target, ModifierCollection modifiers)
| Annotation(IExpression name, ArgumentCollection arguments)
| TypeConstrain()
| PropertyDefinitionPart(AnnotationCollection annotations, ModifierCollection modifiers,
  Block body)
| VariableDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
  Type type, Identifier name, IExpression initialValue, ArgumentCollection arguments,
  IntegerLiteral bitField, Block body)
| LinqQuery()
| OrderByKey(IExpression expression, bool ascending)
| BinaryOperator(string sign, BinaryOperatorKind kind)
| UnaryOperator(string sign, UnaryOperatorKind kind)

TypeConstrain
= ValueConstrain(Type type)
| SuperConstrain(Type type)
| ReferenceConstrain(Type type)
| ImplementsConstrain(Type type)
| ExtendConstrain(Type type)
| EigenConstrain(Type type)
| ConstructorConstrain(Type type)

ElementCollection
= AnnotationCollection(IList<Annotation> elements)
| ArgumentCollection(IList<Argument> elements)
| CaseCollection(IList <Case> elements)
| CatchCollection(IList <Catch> elements)
| ExpressionCollection(IList<Expression> elements)
| GenericArgumentCollection(IList<GenericArgument> elements)
| GenericParameterCollection(IList<GenericParameter> elements)
| IdentifierCollection(IList<Identifier> elements)
| ModifierCollection(IList<Modifier> elements)
| OrderByKeyCollection(IList<OrderByKey> elements)
| ParameterCollection(IList<Parameter> elements)
| TypeCollection(IList <Type> elements)
| TypeConstrainCollection(IList<TypeConstrain> elements)
ClassLikeDefinition
= AnnotationDefinition()
| ClassDefinition()
| EigenClassDefinition()
| EnumDefinition()
| InterfaceDefinition()
| ModuleDefinition()
| NamespaceDefinition()
| StructDefinition()
| UnionDefinition()
```

Figure 5-12: Full definition of unified code model in extended ASDL (Part1)

```
Expression
= Call(IExpression target, ArgumentCollection args,
  GenericArgumentCollection genericArguments, Proc proc)
| Cast(Type type, IExpression createExpression)
| Indexer(IExpression current, ArgumentCollection create)
| KeyValue(IExpression key, IExpression value)
| Label(string name)
| New(IExpression target, ArgumentCollection arguments,
  GenericArgumentCollection genericArguments, ArrayLiteral initialValues, Block body)
| Property(string delimiter, IExpression owner, IExpression name)
| Slice(IExpression initializer, IExpression condition, IExpression step)
| Switch(IExpression value, CaseCollection cases)
| Catch(TypeCollection types, IExpression assign, Block body,
  AnnotationCollection annotations, ModifierCollection modifiers)
| If(IExpression condition, Block body, Block falseBody)
| Lambda(Identifier name, ParameterCollection parameters, Block body)
| Proc(ParameterCollection parameters, Block body)
| Try(Block body, CatchCollection catches, Block elseBody, Block finallyBody)
| ConstructorLike<TSelf>(Block body, AnnotationCollection annotations,
  ModifierCollection modifiers, ParameterCollection parameters,
  GenericParameterCollection genericParameters, TypeCollection throws)
| DoWhile(IExpression condition, Block body, Block falseBody)
| For(IExpression initializer, IExpression condition, IExpression step, Block body)
| Foreach(IExpression element, IExpression set, Block body, Block elseBody)
| While(IExpression condition, Block body, Block elseBody)
| Fix(IExpression value, Block body)
| Synchronized(IExpression value, Block body)
| Using(ExpressionCollection expressions, Block body)
| With(IExpression value, Block body)
| ComprehensionBase()
| MapComprehension(KeyValue element, ExpressionCollection generator)
| ClassLikeDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
  IExpression name, GenericParameterCollection genericParameters,
  TypeConstrainCollection constrains, Block body)
| EventDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
  Type type, Identifier name, ParameterCollection parameters,
  PropertyDefinitionPart adder, PropertyDefinitionPart remover)
| FunctionDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
  Type type, GenericParameterCollection genericParameters,
  Identifier name, ParameterCollection parameters, TypeCollection throws,
  Block body, IExpression annotationExpression)
| PropertyDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
  Type type, Identifier name, ParameterCollection parameters,
  PropertyDefinitionPart getter, PropertyDefinitionPart setter)
| Identifier(string name)
| Break(IExpression value)
| Continue(IExpression value)
| Goto(Identifier value)
| Redo()
| Retry()
| Return(IExpression value)
| Throw(IExpression value, IExpression data, IExpression trace)
| YieldBreak(IExpression value)
| YieldReturn(IExpression value)
| LinqExpression()
| ArrayLiteral()
| IterableLiteral()
| ListLiteral()
| MapLiteral()
| SetLiteral()
| TupleLiteral()
| Literal()
| Range(IExpression min, IExpression max)
```

Figure 5-13: Full definition of unified code model in extended ASDL (Part2)

```
| BinaryExpression(IExpression leftHandSide, BinaryOperator binaryOperator,
  IExpression rightHandSide)
| TernaryExpression(IExpression condition, IExpression trueExpression,
  IExpression falseExpression)
| UnaryExpression(IExpression operand, UnaryOperator unaryOperator)
| Alias(IExpression value, IExpression alias)
| Assert(IExpression value, IExpression message)
| Default(Type type)
| Defined(IExpression value)
| Delete(IExpression value)
| Exec(IExpression value)
| Import(IExpression name, string alias, IExpression member, ModifierCollection modifiers)
| Pass(IExpression value)
| Print(IExpression value)
| PrintChevron(IExpression value)
| Sizeof(IExpression expression)
| StringConversion(IExpression value)
| Typeof(IExpression type)
| Type(IExpression basicExpression)
| Block(IList<IUnifiedExpress> elements)
| VariableDefinitionList(IList<VariableDefinition> elements)

ConstructorLike
= Constructor() | InstanceInitializer() | StaticInitializer()

ComprehensionBase
= IterableComprehension(IExpression element, ExpressionCollection generator)
| ListComprehension(IExpression element, ExpressionCollection generator)
| SetComprehension(IExpression element, ExpressionCollection generator)

Identifier
= LabelIdentifier(string name)    | SuperIdentifier(string name)
| ThisIdentifier(string name)     | TypeIdentifier(string name)
| VariableIdentifier(string name) | VaueIdentifier(string name)

LinqQuery
= FromQuery(VariableIdentifier receiver, IExpression source, Type receiverType)
| GroupByQuery(IExpression element, IExpression key, VariableIdentifier receiver)
| JoinQuery(VariableIdentifier receiver, IExpression joinSource,
  IExpression firstEqualsKey, IExpression secondEqualsKey)
| LetQuery(VariableIdentifier variable, IExpression expression)
| OrderByQuery(OrderByKeyCollection keys)
| SelectQuery(IExpression expression, VariableIdentifier receiver)
| WhereQuery(IExpression condition)

Literal
= NullLiteral() | TypedLiteral()

TypedLiteral
= IntegerLiteral(BigInteger value)
| BooleanLiteral(bool value) | CharLiteral(string value)
| StringLiteral(string value) | SymbolLiteral(string value)
| FractionLiteral(double value, FractionLiteralKind kind)
| RegularExpressionLiteral(string value, string options)

IntegerLiteral
= BigIntLiteral() | Int16Literal() | Int31Literal() | Int32Literal()
| Int64Literal() | Int8Literal() | UInt16Literal() | UInt31Literal()
| UInt32Literal() | UInt64Literal() | UInt8Literal()

Type
= BasicType() | WrapType(Type type)

WrapType
= ArrayType() | ConstType() | GenericType() | PointerType()
| ReferenceType() | StructType() | UnionType() | VolatileType()
```

Figure 5-14: Full definition of unified code model in extended ASDL (Part3)

Chapter 6

Case Study of UNICOEN: A Language-Independent Aspect-Oriented Programming Framework

6.1 Introduction of this chapter

In programming, cross-cutting concerns may appear in many modules of software. Since concerns, such as logs and the caching process, are scattered on many modules and are interweaved with other core concerns, the outlook of the code will deteriorate with repeated modification and the oblivescences of modifying or erasing the portions of concern will increase. This will lead to the reduced maintainability of software. Thus, the separation of cross-cutting concerns and other core concerns is one of the challenges in programming.

Aspect-oriented programming (AOP) has been proposed as a means of solving this problem [47]. In AOP, cross-cutting concerns are written separately from core concerns, and they are woven into core concerns later. In addition, when there are multiple cross-cutting concerns, each cross-cutting concern would be modularized into different modules. Therefore, since AOP achieves the separation of cross-cutting concerns, the maintainability of the software does not deteriorate.

Various AOP tools have been proposed. For example, AspectJ [49] is an extension of Java language, and AOJS [80] supports AOP for JavaScript. Developers can benefit from AOP in most major languages. However, since most existing AOP tools are implemented for a specific language, these tools cannot deal with cross-cutting concerns scattered on many modules implemented in multiple languages. Therefore, there is a possible that a single concern is not always modularized to a single aspect. Moreover, the weaving mechanism and the description of the aspect are not unified among existing AOP tools. It leads to cost in terms of the time required for learning.

In this chapter, I propose a language-independent AOP framework named UniAspect. I and my developer team developed UniAspect using UNICOEN. UniAspect achieves language independence by translating programs written in various languages into a Unified Code Object (UCO), which is my common representation of source code, and weaving aspects through the UCO.

The Contributions of the chapter are as follows:

- I show cross-cutting concerns that are written in multiple languages and scattered on many modules can be modularized into a single module using UniAspect.

- I explain the learning cost due to the introduction of AOP for a new language can be reduced because UniAspect supports AOP in multiple languages.

UniAspect is ongoing project as open-source software, and it can be downloaded from the UniAspect website [65].

In this chapter, I introduce UniAspect as follows. Section 2 illustrates the problems in existing AOP tools. Sections 3 and 4 give an overview of UniAspect and the UCO, respectively. I describe the joinpoint model and the implementation of UniAspect in Sections 5 and 6, respectively. Section 7 reports a case study. Section 8 refers to related work and Section 9 concludes the chapter.

6.2 Background

In this section, I illustrate the problems in existing AOP tools. These are as follows.

P1: Most existing AOP tools cannot deal with cross-cutting concerns, which are scattered on many modules implemented in two or more languages.

For example, web applications are usually implemented by using multiple languages because the client side program and the server side program run on the different platform. To obtain logs of such a web application, it is necessary to use two AOP tools to support the implementation languages on the client side and server side (Figure 6-1). In this example, AOJS is used for the client side and AspectJ is used for the server side. Therefore, the log code must be written as two aspects, as shown in Figures 6-2 and 6-3, and it is difficult to deal with these aspects as a single module. As a result, if the module has been modified about a concern, it is necessary to confirm its extent; how many languages are affected. Incorrect confirmation leads to misses of the modification.

P2: There is no consistency in the weaving mechanism and the description of the aspect among existing AOP tools.

Each existing AOP tool has its own mechanism for weaving. Therefore, developers need to pay a lot of attention to the consistency of weaving among multiple tools. In addition, the description of the aspect varies depending on the tool: an extended grammar of the specific language (Figure 6-3), a XML notation (Figure 6-2) and a function provided by AOP library within specific language's grammar [79][9]. Thus, the introduction of a new AOP tool results in a cost in terms of the time required for learning.

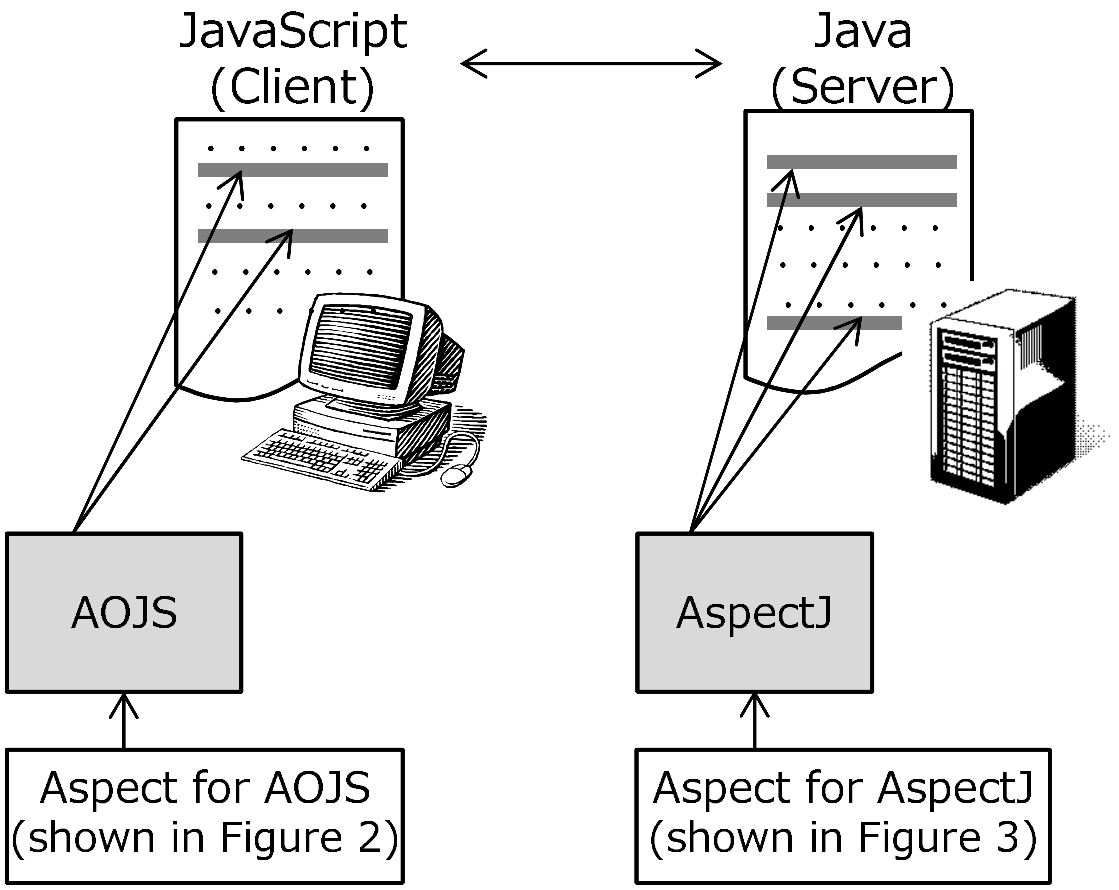

JavaScript (Client)
Java (Server)
AOJS
AspectJ
Aspect for AOJS (shown in Figure 2)
Aspect for AspectJ (shown in Figure 3)

Figure 6-1: Aspects for Web application

```
<?xml version="1.0" ?>
<aspectsetting>
  <function functionname="/*"
        pointcut="execution">
    <before><![CDATA[
        console.log(__name__ +
            " is executed."); ]]>
    </before>
  </function>
</aspectsetting>
```

Figure 6-2: Aspect example in AOJS

6.3 Overview of UniAspect

In this section, I give an overview of UniAspect. UniAspect translates programs written in various languages into a UCO and weaves aspects through the UCO. In particular, UniAspect is designed to deal with only common elements among various

```
public aspect Logger {
  pointcut allMethod() :
      execution(* *.*());

  before() : allMethod() {
    System.out.println(
      thisJoinPoint.getSignature()
      + " is executed.");
  }
}
```

Figure 6-3: Aspect example in AspectJ

programming languages such as function call, making it possible to specify the join-point (particular point in the program) in the unified aspect description, regardless of the language. This allows developers to implement cross-cutting concerns in multiple languages as a single aspect (Figure 6-4). In other words, this allows developers to deal with only a single module when they modify a concern. UniAspect can also be used as an AOP tool for multiple languages. The details of the UCO will be described, in section 4.

Figure 6-5 shows an overview of UniAspect. The entire process of the system is as follows, where numbers correspond to those in the Figure. Each module of the system will be explained in detail, in section 6.

1. The developer inputs source codes written in supported language and aspect.

2. Weaving information and code fragments from the aspect are extracted in the aspect analyzer.

3. The UCOs of the input source code and code fragments are generated in the UCO generator.

4. Code fragments are woven into the input source code in the UCO in the weaver.

5. The source code with aspect is regenerated from the UCO and outputted.

UniAspect regenerates a source code after weaving aspects through the UCO generated from the input source code. In other words, UniAspect performs weaving by transformation of the source code on the UCO. Therefore, developers can compile or execute source code with aspect using any system.

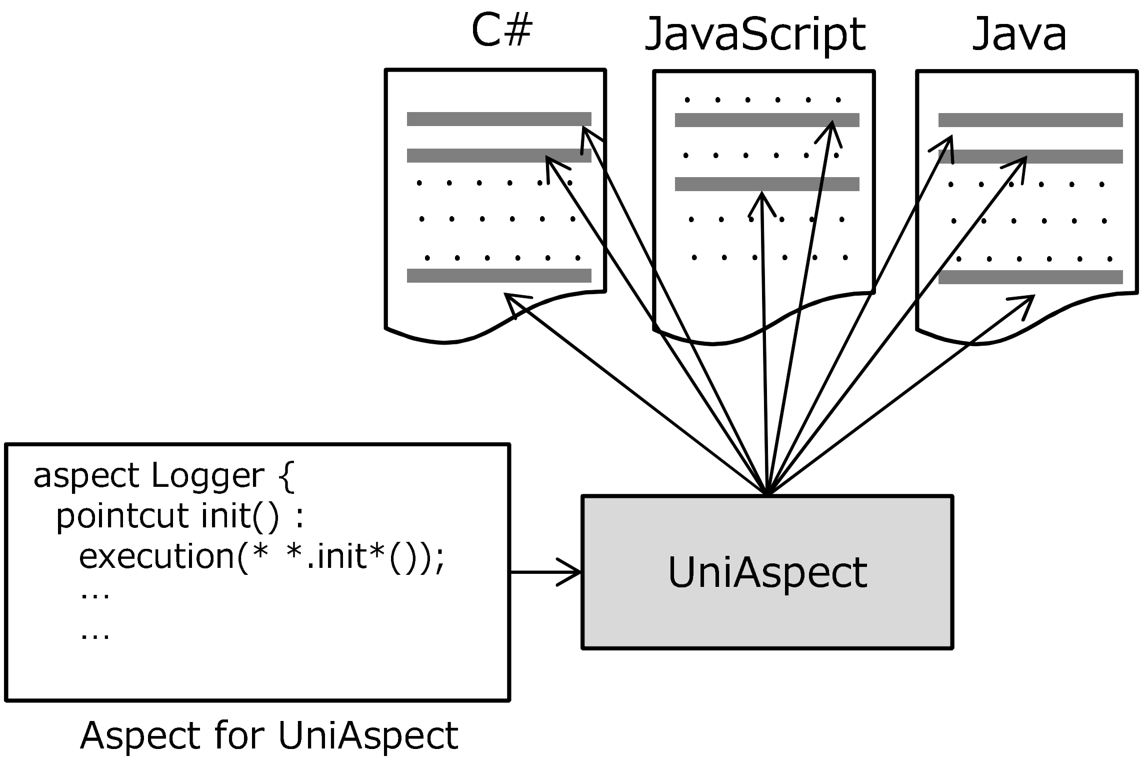


Figure 6-4: Aspect weaving by UniAspect

6.4 Unified Code Object

UniAspect is based on UNICOEN [72], a source code processing framework for multiple languages. UNICOEN is a framework for developing source code analysis or transformation tools, and my research team is developing it as open-source software. UNICOEN supports the development of language-independent source code processing tools by supplying a common representation of source code for different programming languages. The common representation that UNICOEN supplies is called the unified code model (UCM), and objects generated from the source code according to the UCM are called UCOs.

For example, an If statement is composed of a conditional expression, a true block and a false block in all programming languages. Therefore, UNICOEN translates an If statement as shown in Figure 6-6. UNICOEN also translates elements that appear in particular languages such as Type and Class into a common object. As a result, a

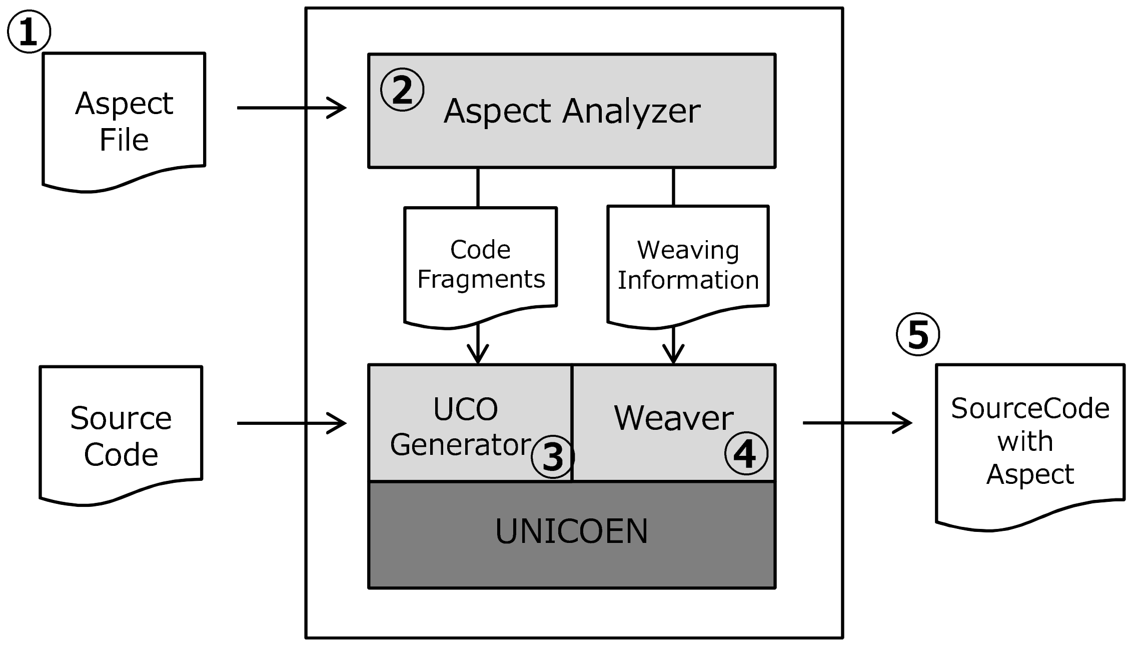


Figure 6-5: Overview of UniAspect

UCM consists of the sum of a set of elements in programming languages as shown in Figure 6-7. The latest version of UNICOEN deals with seven programming languages, C, Java, C#, JavaScript, Ruby, Python and Visual Basic, and a UCM consists from these seven languages. Developers are also able to implement new languages into UNICOEN by writing mapping rules for the UCO in UCM.

6.5 Joinpoint Model

In this section, I present an example of an aspect in UniAspect and explain its features. Before the explanation, I define some terms used in this section.

Aspect: A module that includes a pointcut, advice and an intertype declaration.

Joinpoint: A location that is specified as a weaving point.

Pointcut: A condition for selecting a weaving point from a set of joinpoints.

Advice: A code fragment that will be woven into a specified joinpoint.

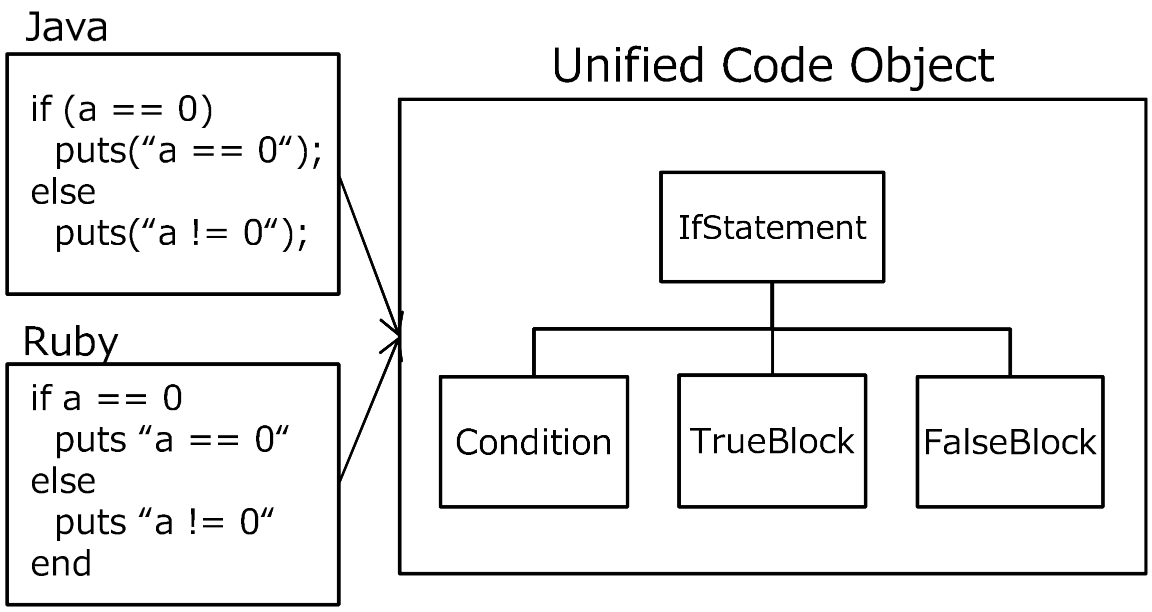


Figure 6-6: Example of unified code object

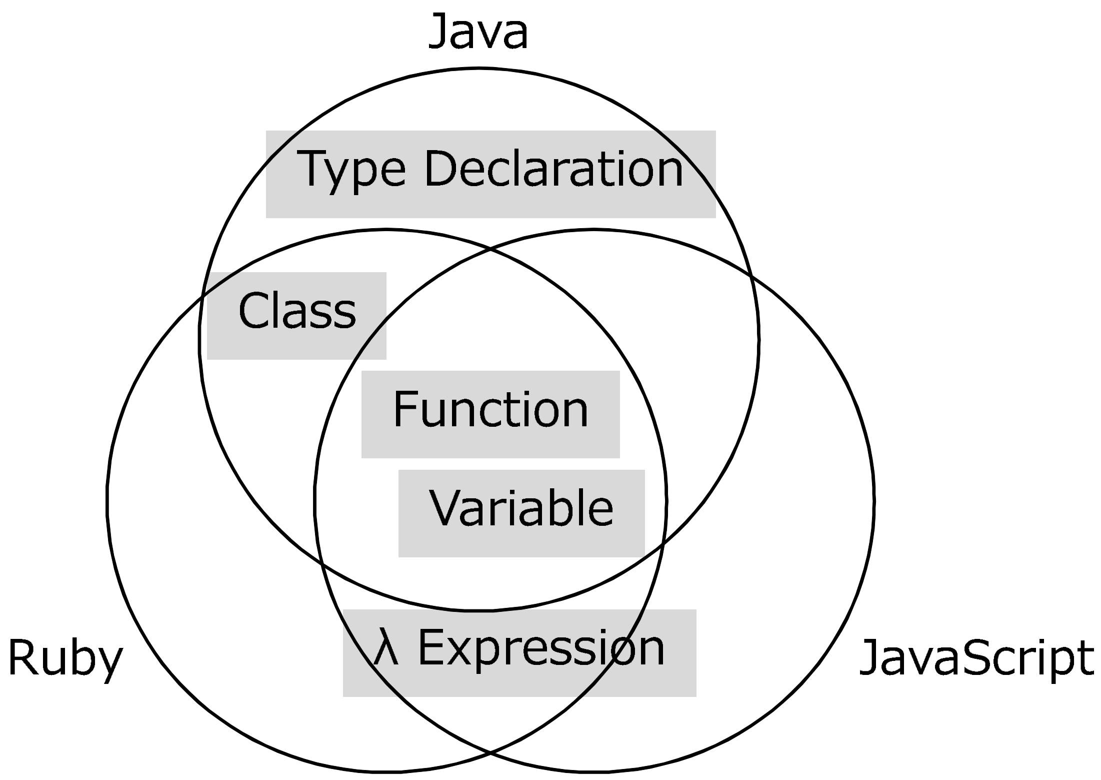


Figure 6-7: Set of elements in unified code model

Intertype declaration: A mechanism used to add functions and variables statically into a target class or file.

6.5.1 Sample Program

Figure 6-8 shows an example of an aspect in UniAspect. It is designed to imitate a grammar of AspectJ so that a developer using an existing system can understand it easily. There is a problem that code fragments written in advice and intertype declarations depend on the programming language of the weaving target. Since UNICOEN does not perform a semantic analysis of the source code, it does not deal with the feature such as language translation. For example, weaving advice written in Java into a source code written in C# will result in an error. Therefore, UniAspect does not support a common description of advice and intertype declarations. As a result, an aspect in UniAspect has both a language-independent portion and a language-dependent portion. A process that depends on a particular language is called a language-dependent block and is described as shown in lines 15 to 17 of Figure 6-8. A language-dependent block is used to describe advice and intertype declarations. The remainder of this section explains details of an aspect in UniAspect.

```
 1  aspect Sample {
 2    Foo : @Java{
 3      private int x = 10;
 4      public void __debug() {
 5        System.out.println(...);
 6      }
 7    }end
 8
 9    pointcut move() :
10      execution(double Foo.hoge());
11    pointcut init() :
12      execution(* *.init*());
13
14    before : init() {
15      @Java {
16        __debug();
17      }end
18      @JavaScript {
19        console.log(JOINPOINT_NAME +
20            " is executed.");
21      }end
22    }
23  }
```

Figure 6-8: Example of aspect in UniAspect

6.5.2 Pointcut

To weave an aspect into programs written in supported language, UniAspect is designed to deal with only common elements as joinpoints in a UCO among multiple languages. Table 6.1 shows the relationship between programming languages and the main elements in a UCO. "Yes" indicates that the element appears in the corresponding programming language, and "No" indicates that the element does not appear.

Table 6.1: Relationship between languages and main elements in the unified code object

	C	Java	C#	JavaScript	Ruby	Python	VisualBasic
Function Declaration	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Function Call	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Variable	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Exception	No	Yes	Yes	Yes	Yes	Yes	Yes
Pointer	Yes	No	Yes	No	No	No	Yes
Class	No	Yes	Yes	No	Yes	Yes	Yes
Typed Variable Declaration	Yes	Yes	Yes	No	No	No	Yes

Table 6.1 shows that function declarations, function calls and variables appear in all programming languages adopted by UNICOEN. Moreover, variables are categorized into two types, assignment and reference variables. Table 6.2 shows specifiable joinpoints in UniAspect. Using these joinpoints, for example, developers can implement logs of function performance and variable values as an aspect.

Table 6.2: List of specifiable joinpoints

Joinpoint Type	Location in Source Code
call	When a function is called
execution	When a function is executed
get	When a variable is referenced
set	When a variable is assigned

A pointcut is declared to use the "pointcut" identifier shown in lines 9 and 11 of Figure 6-8, and a developer writes the pointcut name and the conditions for selecting joinpoints. In the conditions, developers can specify the type of joinpoint, the return type, the class name and the function/variable name. Developers can specify each parameter by directly giving a string name and also by using wildcards. Wildcards match any string in a class name, function name and so forth.

For example, lines 9 and 10 of Figure 6-8 show the declaration of a pointcut that specifies function execution whose return type is "double", whose parent class name is "Foo" and whose function name is "hoge".

Because UniAspect weaves aspects through a UCO, a pointcut specifies the joinpoint from the UCO. Therefore, the description of a pointcut is independent of the programming language. For example, lines 11 and 12 of Figure 6-8 select function declarations whose name starts with "init" as a weaving point regardless of the language. However, for example, it is not possible to select a joinpoint by specifying some parameters in a language that does not have a type and class such as JavaScript. Therefore, weaving is only performed when using wildcards. A declared pointcut is referenced from advice such as that in line 14 of Figure 6-8 and is used for identifying weaving points.

6.5.3 Advice

Developers can specify "before" and "after" as advice. For example, to measure the execution time of a function, developers need to implement a measuring code before and after appropriate points. Therefore, it is necessary to specify locations before and after a joinpoint. "Before advice" will be woven into before a joinpoint, and "after advice" will be woven into after a joinpoint.

Advice is declared to use a "before" or "after" identifier, as shown in line 14 of Figure 6-8, and the developer writes the declared pointcut and code fragments. As already mentioned in the section 5.1, because code fragments depend on the programming language of the target to be woven, developers need to write code fragments in language-dependent blocks in the appropriate programming language.

For example, lines 15 to 21 of Figure 6-8 define language-dependent blocks for Java and JavaScript. If a joinpoint selected from the corresponding pointcut is based on a Java program, the code shown in line 16 of Figure 6-8 will be woven, and if it is based on a JavaScript program, the codes shown in lines 19 and 20 of Figure 6-8 will be woven.

If advice refers to a pointcut that selects multiple joinpoints, developers are required to obtain information about the selected joinpoints to describe advice flexibly. For example, when a developer wants to output logs that show which functions have been executed, it is necessary to obtain the function names. Therefore, UniAspect is designed to enable developers to refer to joinpoint information. For example, for lines 19 and 20 of Figure 6-8, if the function name of a joinpoint is "initRequest", the output string will be "initRequest is executed".

6.5.4 Intertype Declaration

If a developer needs a new function or variable in advice, the necessary function or variable can be defined using an intertype declaration. Lines 2 to 7 of Figure 6-8 define an intettype declaration by describing a language-dependent block. The developer specifies the class name by describing an identifier before the language-dependent block in the case of a language with classes such as Java, or specifies the file name in the case of a language that does not have declarative classes such as JavaScript. Then, the corresponding function or variable will be woven into the specified class or file.

On the other hand, UniAspect does not produce its own scope when weaving the both advice and intertype declarations. Since UniAspect realizes aspect weaving by transformation of source code, advice and intertype declarations are woven into directly specified joinpoints. As a result, the scope depends on the joinpoint used for weaving. Therefore, it is possible to refer to a function and variable that belong to the scope of a joinpoint used for weaving. On the other hand, it is possible for a conflict to arise between variable/function definitions of aspect and those of target program. Regarding the scope, UniAspect guarantees safe weaving in terms of syntax, as described in section 6.3, but does not guarantee safety in terms of semantics. Thus, developers need to try to avoid interference between aspects and source code.

6.5.5 Association of Aspects and Source Code

Developers need to specify the aspect file and directory including source codes as an input. UniAspect recursively searches for files in the specified directory, and if an appropriate file that is written in a supported programming language is found, weaving process is performed. The result of weaving is outputted as source codes with the hierarchy of the original directories, and files that are not subjected to weaving are outputted without change. Finally, developers can (compile and) execute output files using any system.

6.6 Implementation

In this section, I provide details of the implementation of UniAspect. UniAspect consists of three modules, an aspect analyzer, a UCO generator and a weaver, as shown in Figure 6-5. The UCO generator and weaver are based on UNICOEN and deal with the processing of UCOs. The remainder of this section provides details of each module.

6.6.1 Aspect Analyzer

The aspect analyzer receives an aspect file as the input and extracts information on the pointcut, advice and intertype declarations. This module is based on a parser created from ANTLR [66], which is a well-known compiler compiler, and parses aspect files in accordance with the BNF, as shown in Figure 6-9. The result of parsing is expressed as the syntax tree, and this module obtains weaving information by scanning the tree.

Since the code written within a language-dependent block obeys programming language rules, the aspect analyzer accepts any string as the content of a language-dependent block. By doing this, each string within the language-dependent block will

```
aspect
  : 'aspect' IDENTIFIER aspectBody

aspectBody
  : '{' element* '}'

element
  : intertypeDeclaration
  | pointcutDeclaration
  | adviceDeclaration

languageDependentBlock
  : languageDeclaration
    '{' CONTENTS '}end'

languageDeclaration
  : '@' languageType

languageType
  : 'Java' | 'JavaScript' | 'C' | 'VB'
  | 'CSharp' | 'Ruby' | 'Python'

intertypeDeclaration
  : IDENTIFIER ':' languageDependentBlock

pointcutDeclaration
  : 'pointcut' IDENTIFIER '()'
    ':' pointcutDeclarator ';'

pointcutDeclarator
  : pointcutType '(' TYPE
    IDENTIFIER '.' IDENTIFIER '()' ')'

pointcutType
  : 'execution' | 'call'
    | 'get' | 'set'

adviceDeclaration
  : adviceType ':'
    IDENTIFIER '()' adviceBody

adviceType
  : 'before' | 'after

adviceBody
  : '{' languageDependentBlock+ '}'
```

Figure 6-9: BNF for aspect in UniAspect

be extracted and sent to the UCO generator. On the other hand, the remainder of the aspect file will be checked for correct syntax in this module, and when there is a syntax error, it will be reported. The extracted information about weaving will only be sent to the next modules when the aspect analyzer can successfully parse the file.

6.6.2 UCO Generator

The UCO generator translates the input source code and code fragments extracted from the previous module into a UCO using UNICOEN.

Languages that UniAspect supports are dependent on that of UNICOEN. As described in section 4, currently UNICOEN supports seven languages, and therefore, UniAspect supports same seven languages for AOP. Moreover, since developers are able to implement new languages into UNICOEN by writing mapping rules for the UCO, UniAspect can deal with more languages. Since the weaving process is performed upon the UCO, additional implementation of the weaver is almost not necessary if new language have the joinpoint that have been adopted in UniAspect.

When there is syntax error in the extracted code fragments, it will be reported by this module. The generated UCO is then sent to the weaver.

6.6.3 Weaver

This module weaves a UCO of advice into a UCO of the input source code based on weaving information received from the aspect analyzer. This module performs three steps: identification of the joinpoints, the insertion of advice and the replacement of particular variables. As an example, Figure 6-10 shows the algorithm for weaving into an execution joinpoint. The details of each step are as follows, using the examples in Figures 6-10 and 6-11.

(1) Identification of joinpoints

The weaver identifies joinpoints in a UCO based on received information. Since a UCO forms a tree structure, as shown in Figure 6-11, it is easy to identify specific elements in a UCO of the input source code. For example, since the execution pointcut selects function declarations, this module obtains the list of objects of function declarations (Figure 6-11-1). Line 3 of Figure 6-10 shows how to obtain all function declaration objects, which can be traced from the root object. Other pointcuts behave similarly: a call pointcut gathers all call objects, and a set pointcut gathers all assignment

```
1     ExecutionWeaveing(pointcut, advice){
2       //(1)Identification of joinpoint
3       var functions = root.getAllElements<Function>();
4       foreach(func : functions) {
5         if(func.name == pointcut.name &&
6            func.type == pointcut.type &&
7            func.Parent.name == pointcut.className) {
8
9           //(2)Insertion of advice
10          func.block.insertFirst(advice);
11
12          //(3)Replacement of particular variables
13          var variables = func.getAllElements<Variable>();
14          foreach(variable : variables) {
15            if(variable.name == "JOINPOINT_NAME") {
16              variable.name = func.name;
17            }
18          }
19    } } }
```

Figure 6-10: Algorithm of weaving into execution joinpoint

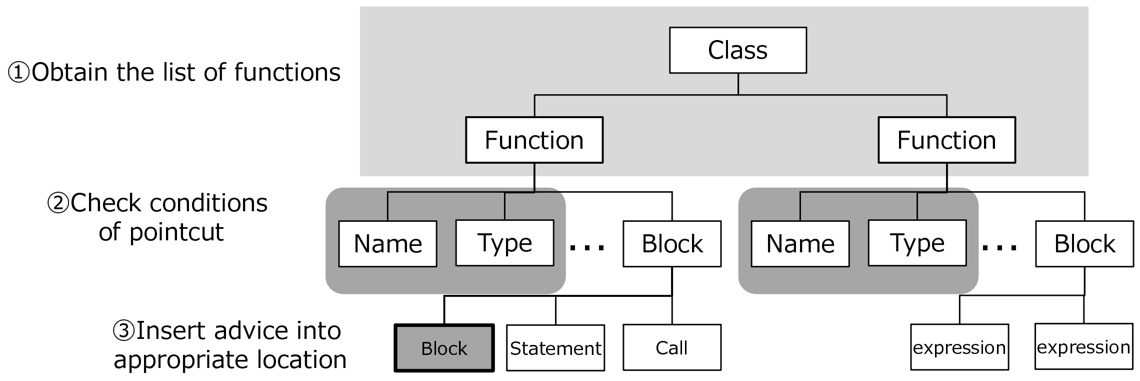

①Obtain the list of functions
②Check conditions of pointcut
③Insert advice into appropriate location
Class
Function
Function
Name
Type
...
Block
Name
Type
...
Block
Block
Statement
Call
expression
expression

Figure 6-11: Process of execution weaving

expression objects.

Next, this module narrows the list of objects in accordance with the given parameters of a pointcut. In the case of an execution pointcut, this module obtains the name, return type and parent name from the linked object, and checks whether they match the given conditions (Figure 6-11-2). Line 5 of Figure 6-10 shows the comparison of the function name and specified name, and lines 6 and 7 of Figure 6-10 show the comparison of the type and belonging class. Finally, the objects that satisfy all given conditions will proceed to the next step.

(2) Insertion of advice

In this step, the weaver inserts the UCO of advice into the joinpoint specified in the previous step. The basic operation is to insert the advice into a block so that it becomes a previous sibling in the case of before (Figure 6-11-3). Line 10 of Figure 6-10 shows how to add an appropriate object of advice into the head of a function block. In the UCO generator, the advice is translated into a UCO as a block, and because the unified code model allows a nesting structure, a block can include another block. UNICOEN can correctly translate this nesting structure into source code. In fact, most programming languages have adopted this syntax in which blocks can include other blocks. For example, Figures 6-12 and 6-13 show examples of nesting Java and JavaScript, respectively. Thus, UniAspect guarantees safe weaving in terms of syntax by inserting advice as a block.

However, in the case of weaving after the execution joinpoint, it is necessary to insert advice in a specific way. For example, if an object is inserted into the end of a block, as shown in Figure 6-14, it will become dead code and an incorrect result will be obtained. Therefore, in this case, a list of all return statements in function block is obtained, and objects are inserted before each return statement. As a result, the advice will always be performed even if a function is ended by any return statement. Alternatively, when a return type of function is void, the advice is inserted into the end of the block depending on the number of return statements in the function.

```
block
  : '{' blockStatement* '}'

blockStatement
  : localVariableDeclarationStatement
  | classOrInterfaceDeclaration
  | statement

statement
  : block
  | ifStatement
  | forStatement
  | (omitted)
```

Figure 6-12: BNF for Java

(3) Replacement of particular variables

```
statementBlock
 : '{' statementList? '}'

statementList
 : statement (statement)*

statement
 : statementBlock
 | ifStatement
 | iterationStatement
 | (omitted)
```

Figure 6-13: BNF for JavaScript

```
int abs(int a) {
  if(a > 0)
    return a;
  else
    return -a;
}
```

Figure 6-14: Function contains return statements

Finally, this module replaces particular variables with joinpoint information as described in section 5.3. Line 13 of Figure 6-10 obtains a list of variables from a UCO and line 15 checks whether their values match "JOINPOINT_NAME". Line 16 replaces each variable with the appropriate joinpoint name, concluding this step. The weaver repeats these three steps for all pointcuts specified in aspect file. Finally, it regenerates source code from the UCO as the final process.

6.7 Case Study

In this section, I give an example of implementing logs using UniAspect for JsUnit [75], which is a client-server-type test framework. The client side of JsUnit is implemented in Java and the server side is implemented in JavaScript. JsUnit is a system that performs tests on web browsers and displays the results on the server side. In this case study, I implement log codes for all functions of JsUnit to determine which functions are executed on the server side and which are executed on the client side. The execution environment is as follows.

- **CPU:** Intel Core i5 M430 2.27GHz

- **Memory:** 4GB
- **Server side:** Java 1.6.26
- **Client Side:** Firefox 7.0.1

In this case study, the server program runs on the same machine as the client program. Therefore, I used one machine. The process in this case study is as follows.

1. Obtain the source code of JsUnit from its Github project page[1].
2. Weave the aspect shown in Figure 6-15 into the obtained source code.
3. Compile the woven source code using the supplied Ant build file and execute it.

Figures 6-16 and 6-18 are code fragments of original source code in JsUnit, and Figures 6-17 and 6-19 are code fragments of source code in JsUnit after weaving aspect. Figures 6-20 and 6-21 show the result of execution. It shows logs of the functions executed on both the server side and client side. I can confirm that the log code has been woven using UniASpect, and as a result, the JOINPOINT_NAME variables shown in Figure 6-15 have been replaced with each function name. I can confirm that the replacement of particular variables has been performed correctly.

```
aspect Logger {
  pointcut allMethod() :
    execution(* *.*());

  before : allMethod() {
  @Java {
    System.out.println(
    JOINPOINT_NAME+" is executed.");
  }end
  @JavaScript {
    console.log(
    JOINPOINT_NAME+" is executed.");
  }end
  }
}
```

Figure 6-15: Aspect using in case study

[1]https://github.com/pivotal/jsunit

Next, table 6.3 shows the number of methods, files and lines in JsUnit. Using UniAspect, 786 log codes have been woven into server-side programs and 43 log codes have been woven into client-side programs. In addition, functions are scattered in 137 files written in Java and JavaScript. However, it would be a laborious task to implement these logs one by one, whereas it is possible to effectively implement cross-cutting concerns such as logs using UniAspect.

Table 6.3: Numbers of woven aspect

	Server side	Client side
Language	Java	JavaScript
Lines of Code	7106	2856
Number of Methods	786	43
Number of Files	128	9

Table 6.4 shows the number of set lines of code, i.e., sequential code that contains no other concerns, and the number of modified files. In the case of using UniAspect, developers can summarize cross-cutting concerns scattered on multiple files written in multiple languages as a single aspect. The execution time required to weave the aspect is 4.71 sec (average of 10 measurements). I consider this time to be acceptable for practical use.

```
public void setResultRepository(
    BrowserResultRepository rep) {
  this.browserResultRepository = rep;
  addTestRunListener(
      new BrowserResultLogWriter(rep));
}

protected List<String> servletNames() {
  List<String> result
      = new ArrayList<String>();
  result.add("acceptor");
  result.add("config");
  result.add("displayer");
  result.add("runner");
  return result;
}

public static void registerInstance
    (JsUnitServer server) {
  JsUnitServer.instance = server;
}
```

Figure 6-16: Fragment of Java program in JsUnit before weaving aspect

Table 6.4: Comparison of scattered log codes

	UniAspect	Existing AOP Tools	Without AOP Tools
Number of set lines of code	1	2	829
Number of modified files	1	2	137

```
  public void setResultRepository(
      BrowserResultRepository rep) {
    {
      System.out.println(
          "setResultRepository" +
          " is executed.");
    }
    this.browserResultRepository = rep;
    addTestRunListener(
      new BrowserResultLogWriter(rep));
  }

  protected List<String> servletNames() {
    {
      System.out.println(
          "servletNames" +
          " is executed.");
    }
    List<String> result
        = new ArrayList<String>();
    result.add("acceptor");
    result.add("config");
    result.add("displayer");
    result.add("runner");
    return result;
  }

  public static void registerInstance
      (JsUnitServer server) {
    {
      System.out.println(
          "registerInstance" +
          " is executed.");
    }
    JsUnitServer.instance = server;
  }
```

Figure 6-17: Fragment of Java program in JsUnit after weaving aspect

The considerations obtained from the results are as follows.

Cross-cutting concerns scattered on many modules implemented in two or more languages can be summarized. When weaving aspects using UniAspect, developers do not need to modify existing source codes; they only need to write aspect files as shown in Figure 6-15. Thus, a developer can summarize cross-cutting concerns scattered on many modules implemented in two or more languages as a single aspect, and this shows that UniAspect solve the problem mentioned as P1. On the other hand, in the case of using existing AOP tools, developers must implement two aspects, for AspectJ and AOJS for example. Therefore, the number of scattered log code is two as shown in table 4, and developers have to manage these aspects separately.

```
function assert() {
  JsUnit._validateargs(1, args);
  var value = JsUnit.
    _nonCommentArg(1, 1, args);
  if (typeof(value) != 'boolean') {
    throw new JsUnit
      .AssertionArgumentError(
        'Bad argument to' +
          ' assert(boolean)');
  }
  JsUnit._assert(
    JsUnit._commentArg(1, args),
    value === true,
    'Call to assert(boolean)' +
      'with false');
}

function assertTrue() {
  JsUnit.
    _validateargs(1, args);
  assert(JsUnit._commentArg(1, args),
    JsUnit._nonCommentArg(1, 1, args));
}

function assertFalse() {
  JsUnit._validateargs(1, args);
  var value = JsUnit.
    _nonCommentArg(1, 1, args);

  if (typeof(value) != 'boolean')
      throw new JsUnit
        .AssertionArgumentError(
          'Bad argument to' +
            'assertFalse(boolean)');
  JsUnit._assert(
    JsUnit._commentArg(1, args),
    value === false,
    'Call to assertFalse(boolean)' +
      ' with true');
}
```

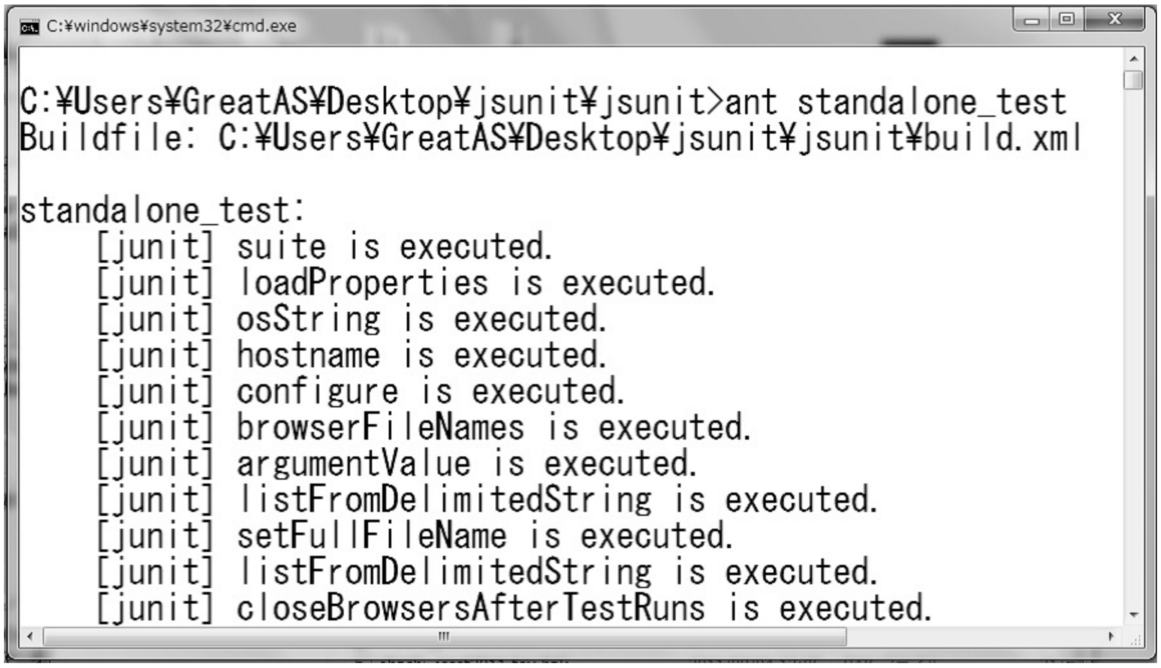
Figure 6-18: Fragment of JavaScript program in JsUnit before weaving aspect

Unified description of aspect. Imagine that existing AOP tools are used for implementing the logs in this case study. For example, when developers use AOJS for JavaScript and AspectJ for Java, the aspect will be described as shown in Figures 6-2 and 6-3, respectively. AspectJ has a similar syntax to Java, but AOJS has a syntax based on XML. Therefore, the use of multiple AOP tools has a large cost including a learning cost. On the other hand, when using UniAspect, these is no cost of introducing an AOP tool to a new language, since the description of the aspect in UniAspect is unified in the form shown in Figure 6-8 or 6-15. This shows that UniAspect solve the problem mentioned as P2.

```
function assert(){
  {
    console.log("assert" +
                " is executed.");
  }
  JsUnit._validateargs(1, args);
  var value = JsUnit.
    _nonCommentArg(1, 1, args);
  if (typeof(value) != 'boolean') {
    throw new JsUnit
      .AssertionArgumentError(
        'Bad argument to' +
          ' assert(boolean)');
  }
  JsUnit._assert(
    JsUnit._commentArg(1, args),
    value === true,
    'Call to assert(boolean)' +
      'with false');
}

function assertTrue(){
  {
    console.log("assertTrue" +
                " is executed.");
  }
  JsUnit.
    _validateargs(1, args);
  assert(JsUnit._commentArg(1, args),
    JsUnit._nonCommentArg(1, 1, args));
}

function assertFalse() {
  {
    console.log("assertFalse" +
                " is executed.");
  }
  JsUnit._validateargs(1, args);
  var value = JsUnit.
    _nonCommentArg(1, 1, args);

  if (typeof(value) != 'boolean')
      throw new JsUnit
        .AssertionArgumentError(
          'Bad argument to' +
            'assertFalse(boolean)');
  JsUnit._assert(
    JsUnit._commentArg(1, args),
    value === false,
    'Call to assertFalse(boolean)' +
      ' with true');
}
```

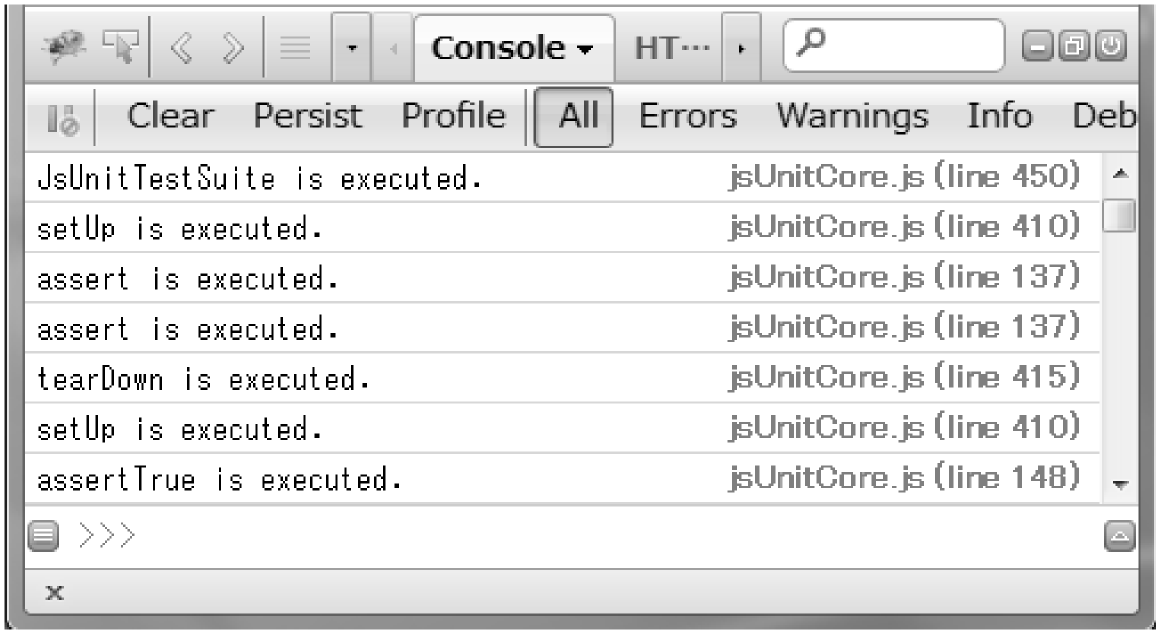
Figure 6-19: Fragment of JavaScript program in JsUnit after weaving aspect

Transformation of source codes. Since the weaving mechanism of UniAspect is based on the transformation of source codes, the compilation of source codes after weaving does not depend on the particular system. On the other hand, existing AOP tools require a dedicated compiler. Therefore, it is possible that developers cannot



```
C:¥Users¥GreatAS¥Desktop¥jsunit¥jsunit>ant standalone_test
Buildfile: C:¥Users¥GreatAS¥Desktop¥jsunit¥jsunit¥build.xml

standalone_test:
    [junit] suite is executed.
    [junit] loadProperties is executed.
    [junit] osString is executed.
    [junit] hostname is executed.
    [junit] configure is executed.
    [junit] browserFileNames is executed.
    [junit] argumentValue is executed.
    [junit] listFromDelimitedString is executed.
    [junit] setFullFileName is executed.
    [junit] listFromDelimitedString is executed.
    [junit] closeBrowsersAfterTestRuns is executed.
```

Figure 6-20: Result of server side logs



```
JsUnitTestSuite is executed.        jsUnitCore.js (line 450)
setUp is executed.                  jsUnitCore.js (line 410)
assert is executed.                 jsUnitCore.js (line 137)
assert is executed.                 jsUnitCore.js (line 137)
tearDown is executed.               jsUnitCore.js (line 415)
setUp is executed.                  jsUnitCore.js (line 410)
assertTrue is executed.             jsUnitCore.js (line 148)
>>>
```

Figure 6-21: Result of client side logs

use an original compiler and the AOP compiler together. However, in the mechanism of transformation of source codes, there is a problem that the aspect may interfere with the source codes as described in section 5.4. Guaranteeing the feasibility of woven source codes is a future work.

Flexibility of aspects. UniAspect accepts the specifications of a joinpoint name directly or using wildcards, but the joinpoint cannot be specified in detail. For example, control flow pointcut, which is adopted in existing AOP tools, or a new joinpoint using the characteristic of the UCO should be considered.

6.8 Related Work

Various AOP tools have been proposed as systems depending on specific languages [49][78][79][9]. These tools provide expressive aspect specification according to the features of base languages. However, these tools cannot deal with cross-cutting concerns scattered on modules written in multiple languages, which is addressed in this chapter. And the weaving mechanism and the description of the aspect vary depending on tools.

On the other hand, several AOP tools for .NET Framework have been proposed to support multiple languages. SourceWeave.NET [38] is a weaver based on source code transformation through the CodeDOM; .NET standard for representing source code as abstract syntax tree. SourceWeave .NET is similar to UniAspect in terms of performing weaving at the source code level. Weave.NET [53] and AspectDNG [28] have adopted a other approach for .NET Framework. These tools weave the aspect into the Common Intermediate Language of the .NET Framework. Although these tools are similar to UniAspect in terms of using a common representation for multiple languages, the languages that they support depend on the .NET Framework. Therefore, because my proposed framework does not depend on a particular platform, UniAspect can support more languages for weaving aspects.

Compose* [17] is a compilation and execution framework for the Composition Filters model [1], which supports multiple languages and platform: .NET platform, Java language and platform and C language. Although Compose* is similar to UniAspect in terms of creating a common structural language model to specify where aspect behavior should be applied, its weaver is implemented depending on a specific target language. On the other hand, the weaving process of UniAspect is also performed based on unified code object, which leads unified implementation of the weaver. However, UniAspect requires language-dependent block to describe advice, the implementation of the unified description of advice is a future work.

XWeaver [29, 68] is an aspect weaver for C/C++ and Java that is based on source code transformation techniques. XWeaver translates base source code into XML format using srcML [58]. The weaving process is then performed on the XML format and its aspect is written in dedicated language for XWeaver; AspectX. Although XWeaver is similar to UniAspect in terms of basing on source code transformation techniques, target languages are limited in C/C++ and Java by srcML. On the other hand, UniAspect supports more various languages; JavaScript, C# and so forth.

Gray has proposed AOP tools based on DMS[3], an existing source code transformation system [31]. Because DMS has an analysis system for multiple languages, his proposed tools make it easy to build AOP tools for multiple languages. However, he has not proposed a mechanism for dealing with multiple languages uniformly.

6.9 Conclusion of this chapter

In this chapter, I proposed a language-independent AOP framework, UniAspect. UniAspect achieves language independence by translating programs written in various languages into a UCO then weaving aspects through the UCO.

As a case study, I gave an example of implementing logs for a system written in Java and JavaScript. Because developers can summarize cross-cutting concerns scattered on modules in multiple languages as a single aspect, UniAspect will contribute to the increased maintainability of codes for obtaining execution logs of series of functionality supplied in multiple languages, for example. Because UniAspect is available as an AOP tool for a single supported language and provides a unified description of the aspect, it will also contribute to decreasing the learning cost of AOP tools.

Finally, I describe some future works.

Language dependence of advice. Although UniAspect provides a unified description of an aspect, advice is dependent on the language of the joinpoint. To achieve a language-independent description of advice, I should provide common specification for describing advice, or programming language translation system for code fragments on advice, which can be also implemented using UNICOEN.

Guarantee of semantics in weaving. UniAspect guarantees safe weaving in terms of syntax as described in section 6.3. However, it does not guarantee the feasibility of the output source code in terms of semantics. It becomes apparent when it is compiled or executed. I should consider the immediately compilation of output code and discard output code that cannot be woven.

Flexible description of aspects. Although UniAspect supports basic pointcuts in its specification, it cannot describe aspect flexibly. For example, UniAspect does not support several pointcut adopted in existing tools; control flow pointcut, composition of pointcuts and so forth. I plan to support these pointcuts to make an aspect more expressive.

Chapter 7

Conclusion

7.1 Summary

Chapter described OCCF, which inserted instrumentation code into source code through ASTs to measure coverage. OCCF extracted the commonalities from the insertion implementation by utilizing the structures of ASTs are similar. As the evaluation, OCCF reduced development and maintenance costs, made flexible measurement, and made complete measurements.

Chapter described a technique of detecting duplicated test code defining the inclusion relation of test code based on coverage. This chapter also described a tool that detected duplicated test code. The tools allow user changing coverage criteria to adjust the detection strictness. As a case study, the technique was applied to a sample program and open source software.

Chapter described UNICOEN for manipulating source code supporting multiple programming languages. UNICOEN provides UCM and the the two APIs for developing tools and adding the new support of programming languages. As a case study, this chapter showed UniMetrics and extension of CodeCity. This chapter also confirmed UNICOEN reduced costs of development tool and programming language supports and reduced differences between tools for different programming languages.

Chapter described a language-independent AOP framework, called UniAspect. UniAspect manipulates source code by manipulating a UCO generated by UNICOEN, and thus, UniAspect provided language-independent aspects. As a case study, a sample aspect using UniAspect written in Java and JavaScript for implementing logs for web applications using Java and JavaScript. UniAspect improved modularity rather than conventional AOP processors because UniAspect modularized cross-cutting concerns scattered on modules in multiple languages as a single aspect.

7.2 Future Work

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

I will improve the usability of the detection tool by displaying graphically duplicated test code and an execution path for duplicated test code in future work. Moreover, I plan to detect duplicated test code in black-box testing by using formalized specifications such as contracts in design-by-contract programming.

Although UNICOEN and does not support functional programming languages such as Haskell and OCaml, I plan to add supports of such programming languages in UNICOEN extending UCM. UNICOEN requires users to implement object-code mappers and to extend UCM to add new programming language supports by hand. Although these works require less efforts than expansion for existing programming language processors, I plan to develop a generator for object-code mappers from a mapping specification and a generator for UCM from a model specification such as ASDL.

Although UniAspect provides a unified description of an aspect, advice is dependent on the language of the joinpoint. To achieve a language-independent description of advice, I will provide common specification for describing advice, or programming language translation system for code fragments on advice, which can be also implemented using UNICOEN.

In future work, I will develop more applications using OCCF and UNICOEN. I plan to develop a testing tool using gamification which encourages testers to improve test quality because conduction testing is sometimes considered as tedious work in the real word. I also plan to develop a mutation testing tool supporting multiple programming languages because existing tools support only few programming languages such as C and Java.

Bibliography

[1] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '92, pages 372–395, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-55668-0. URL http://dl.acm.org/citation.cfm?id=646150.679210. 150

[2] N. Batchelder. coverage 3.5.3 : Python package index. URL http://pypi.python.org/pypi/coverage. 42, 98

[3] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms(r): Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. 115, 151

[4] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321146530. 45, 73

[5] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321278658. 74

[6] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL http://dl.acm.org/citation.cfm?id=998675.999417. 75, 93

[7] Z. Blut. Saikuro : A cyclomatic complexity analyzer. URL http://saikuro.rubyforge.org/. 110

[8] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72 (1-2):52–70, June 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.11.003. 115

[9] A. Bryant and R. Feldt. Aspectr readme. URL http://aspectr.sourceforge.net/. 32, 127, 150

[10] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 197–211, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi: 10.1145/117954.117970. URL http://doi.acm.org/10.1145/117954.117970. 28

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994. ISSN 0098-5589. doi: 10.1109/32.295895. URL http://dx.doi.org/10.1109/32.295895. 28

[12] M. L. Collard, M. J. Decker, and J. I. Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 173–184, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4347-5. doi: 10.1109/SCAM.2011.19. 115

[13] S. F. Conservancy. Simplified wrapper and interface generator. URL http://www.swig.org/. 55

[14] L. Copeland. *A Practitioner's Guide to Software Test Design.* Artech House, Inc., Norwood, MA, USA, 2003. ISBN 158053791X. 23, 40

[15] J. R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61 (3):190–210, Aug. 2006. ISSN 0167-6423. doi: 10.1016/j.scico.2006.04.002. 115

[16] D. Crockford. Jslint,the javascript code quality tool. URL `http://www.jslint.com/`. 95

[17] A. R. de, M. Hendriks, W. Havinga, P. Durr, and L. Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008*, Cyprus, July 2008. No publisher. URL `http://doc.utwente.nl/64984/`. 150

[18] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok. Refactoring test code. Technical report, Amsterdam, The Netherlands, The Netherlands, 2001. 74, 93

[19] E. Doernenburg. erikdoe / pmcs / overview - bitbucket. URL `https://bitbucket.org/erikdoe/pmcs/`. 110

[20] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, Oct. 2001. ISSN 0001-0782. doi: 10.1145/383845.383853. URL `http://doi.acm.org/10.1145/383845.383853`. 21

[21] J. Erdfelt, J. Lewis, G. Lukasik, J. Mares, and J. Thomerson. Cobertura. URL `http://cobertura.sourceforge.net/`. 42

[22] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997. ISSN 0001-0782. doi: 10.1145/262793.262798. URL `http://doi.acm.org/10.1145/262793.262798`. 41

[23] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998. ISBN 0534954251. 21

[24] A. S. Foundation. Apache commons bcel. URL `http://jakarta.apache.org/bcel/`. 62

[25] Free Software Foundation, Inc. Gcc, the gnu compiler collection - gnu project - free software foundation (fsf). URL http://gcc.gnu.org/. 42

[26] N. Fukuyasu, S. Yamamoto, and K. Agusa. Case tool platform sapid based on a fine grained repository(special issue on parallel processing). *IPSJ Journal*, 39 (6):1990–1998, 1998-06-15. ISSN 03875806. 115

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. 51, 54

[28] T. Gil. Aspectdng. URL http://aspectdng.tigris.org/. 150

[29] P. S. GmbH. Xweaver. URL http://www.pnp-software.com/XWeaver/. 151

[30] gnoso. Ncover: .net code coverage for .net developers. URL http://www.ncover.com/. 42

[31] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 36–45, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. doi: 10.1145/976270.976277. URL http://doi.acm.org/10.1145/976270.976277. 151

[32] O. M. Group. Omg unified modeling language (omg uml) infrastructure version 2.3. Technical Report formal/2010-05-03, OMG, 2010. URL http://www.omg.org/spec/UML/2.3/Infrastructure/PDF. 51

[33] U. Hafner. Task scanner plug-in. URL http://wiki.hudson-ci.org/display/HUDSON/Task+Scanner+Plugin. 89

[34] K. T. K. S. T. N. Haruhiko Okumura, Houki Satoh. *Algorithm cyclopedia in Java*. Gijutsu-Hyohron, 2003. 60, 69

[35] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue. A pluggable tool for measuring software metrics from source code. In *Proceedings of*

the 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, IWSM-MENSURA '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4565-3. 115

[36] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, December 2004. ISSN 0362-1340. 95

[37] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X. 74

[38] A. Jackson and S. Clarke. Sourceweave.net: Cross-language aspect-oriented programming. In *In the Proceedings of Generative Programming and Component Engineering: Third International Conference (GPCE 2004*, pages 24–28, 2004. 150

[39] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978. 97

[40] C. Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0-201-48542-7. 33

[41] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL `http://doi.acm.org/10.1145/1101908.1101949`. 35

[42] T. Kamiya. the archive of ccfinder official site. URL `http://www.ccfinder.net/`. 75

[43] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*,

28(7):654–670, July 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019480. URL http://dx.doi.org/10.1109/TSE.2002.1019480. 92

[44] S. Karus and H. Gall. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 13–22, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985447. 33, 98

[45] E. G. Kent Beck and D. Saff. Reik schatz. URL http://wiki.hudson-ci.org/display/HUDSON/Testability+Explorer+Plugin. 89

[46] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es), Dec. 1996. ISSN 0360-0300. doi: 10.1145/242224.242420. URL http://doi.acm.org/10.1145/242224.242420. 21

[47] G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 313–, New York, NY, USA, 2001. ACM. ISBN 1-58113-390-1. doi: 10.1145/503209.503260. URL http://doi.acm.org/10.1145/503209.503260. 125

[48] G. Kiczales and E. Hilsdale. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5):313–, Sept. 2001. ISSN 0163-5948. doi: 10.1145/503271.503260. URL http://doi.acm.org/10.1145/503271.503260. 21

[49] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4. URL http://dl.acm.org/citation.cfm?id=646158.680006. 32, 95, 126, 150

[50] B. Kinnersley. The language list. URL http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm. 22

[51] T. Kiri, T. Miyoshi, S. Kishigami, T. Osato, and T. Sonehara. About the source code insertion type coverage tool. In *The 69th Information Processing Society of Japan National Convention*, 2003. 70

[52] B. Kitchenham. What's up with software metrics? - a preliminary mapping study. *J. Syst. Softw.*, 83(1):37–51, Jan. 2010. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.041. URL `http://dx.doi.org/10.1016/j.jss.2009.06.041`. 21

[53] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 1–12, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. 150

[54] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`. 114

[55] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Trans. Softw. Eng.*, 13(6):697–708, June 1987. ISSN 0098-5589. doi: 10.1109/TSE.1987.233475. URL `http://dx.doi.org/10.1109/TSE.1987.233475`. 21

[56] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 52 –60, may 1993. doi: 10.1109/METRIC.1993.263801. 28

[57] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390648. URL `http://doi.acm.org/10.1145/1390630.1390648`. 28

[58] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 289–292, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1495-2. URL http://dl.acm.org/citation.cfm?id=580131.856986. 151

[59] T. J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. 28

[60] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. ISBN 0131495054. 74

[61] Microsoft. Dynamic language runtime, . URL http://dlr.codeplex.com/. 50

[62] Microsoft. Managed extensibility framework, . URL http://www.codeplex.com/MEF/. 50

[63] Microsoft. Visual c++ resources, . URL http://msdn.microsoft.com/en-us/vstudio/hh386302.aspx. 67

[64] A. Monden, S. Sato, T. Kamiya, and K. Matsumoto. Analyzing the quality of legacy software based on code clone. In *IPSJ Journal*, pages 2178–2188, 2003. 74

[65] A. Ohashi. Uniaspect website in unicoenproject. URL http://www.unicoen.net/application/uniaspect.html. 126

[66] T. Parr. Antlr parser generator. URL http://www.antlr.org/. 51, 137

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

ACM. ISBN 1-59593-042-6. doi: 10.1145/1052898.1052914. URL http://doi.acm.org/10.1145/1052898.1052914. 70

[68] O. Rohlik, A. Pasetti, P. Chevalley, and I. Birrer. An Aspect Weaver for Qualifiable Applications. In *Data System in Aerospace (DASIA) Conference*, Nice, July 2004. URL http://control.ee.ethz.ch/index.cgi?page=publications;action=details;id=2129. 151

[69] V. Roubtsov. Emma: a free java code coverage tool. URL http://emma.sourceforge.net/. 27, 42, 98

[70] K. Sakamoto, A. Ohashi, D. Ota, H. Iwasawa, and T. Kamiya. Unicoenproject/unicoen - github, . URL https://github.com/UnicoenProject/UNICOEN. 99

[71] K. Sakamoto, K. Shimojo, and R. Takasawa. Opencodecoverageframework, . URL https://github.com/exKAZUu/OpenCodeCoverageFramework. 41

[72] K. Sakamoto, A. Ohashi, D. Ota, H. Washizaki, and Y. Fukazawa. Unicoen: A unified framework for code engineering supporting multiple programming languages. In *IPSJ/SIGSE Software Engineering Symposium*, volume 2012, pages 1–8, aug 2012. 130

[73] Y. Sakata, K. Yokoyama, H. Washizaki, and Y. Fukazawa. A precise estimation technique for test coverage of components in object-oriented frameworks. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, APSEC '06, pages 11–18, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2685-3. doi: 10.1109/APSEC.2006.11. URL http://dx.doi.org/10.1109/APSEC.2006.11. 45

[74] Y. Sano, Y. Higo, and S. Kusumoto. Comparison of fixing frequency between duplicate code and non-duplicate code. In *IEICE Tech. Rep.*, pages 43–48, 2010. 74

[75] J. Schaible. Jsunit: Jsunit. URL http://jsunit.berlios.de/. 142

[76] R. Schatz. Hudson team foundation server plug-in. URL http://wiki.hudson-ci.org/display/HUDSON/Team+Foundation+Server+Plugin. 89

[77] SonarSource. Sonar. URL http://www.sonarsource.org/. 110

[78] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. ISBN 0-909925-88-7. URL http://dl.acm.org/citation.cfm?id=564092.564100. 32, 150

[79] R. Toledo, P. Leger, and E. Tanter. Aspectscript: expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 13–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-958-9. doi: 10.1145/1739230.1739233. URL http://doi.acm.org/10.1145/1739230.1739233. 127, 150

[80] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. Aojs: aspect-oriented javascript programming framework for web development. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '09, pages 31–36, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-450-8. 32, 95, 126

[81] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. 110

[82] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 196–205, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2131-2. doi: 10.1109/ASE.2004.61. URL http://dx.doi.org/10.1109/ASE.2004.61. 75

[83] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM. ISBN 1-59593-408-1. doi: 10.1145/1138929.1138949. URL http://doi.acm.org/10.1145/1138929.1138949. 27

[84] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *Comput. J.*, 52(5):589–597, 2009. 27

[85] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Trans. Softw. Eng.*, 32:240–253, April 2006. ISSN 0098-5589. 95

[86] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL http://doi.acm.org/10.1145/267580.267590. 21

List of Publications

Journal Papers

- ○ **UNICOEN: A Unified Framework for Code Engineering Supporting Multiple Programming Languages**
  Kazunori Sakamoto, Akira Ohashi, Daichi Ota, Hironori Washizaki and Yoshiaki Fukazawa
  Journal of Information Processing Society of Japan (情報処理学会論文誌), Vol.54, No.2, 2013. (accepted, in Japanese)
- **A Framework for Game Software Which Users Play through Artificial Intelligence Programming**
  Kazunori Sakamoto, Akira Ohashi, Hironori Washizaki and Yoshiaki Fukazawa
  IEICE Transactions on Information and Systems (電子情報通信学会論文誌), Vol.J95-D, No.3 pp.412-424, 2012. (in Japanese)
- **Software Engineering Approaches for Design Patterns**
  Hironori Washizaki, Kazunori Sakamoto, Naoki Ohsugi, Katsuhiko Gondow, Satoshi Hattori, Atsuto Kubo, Takashi Kobayashi, Mika Ohtsuki, Katsuhisa Maruyama, Akira Sakakibara
  Computer Software - JSSST Journal (コンピュータソフトウェア), Vol.29, No.1, pp.130-146, 2012. (in Japanese)
- **Open Code Coverage Framework: A Framework for Consistent, Flexible and Complete Measurement of Test Coverage Supporting Multiple Programming Languages**
  Kazunori Sakamoto, Fuyuki Ishikawa, Hironori Washizaki and Yoshiaki Fukazawa
  IEICE Transactions on Information and Systems, Vol.E94-D, No.12, pp.2418-2430, 2011.

International Conference Presentations

- ○ **OCCF: A Framework for Developing Test Coverage Measurement Tools Supporting Multiple Programming Languages**
  Kazunori Sakamoto, Kiyofumi Shimojo, Ryohei Takasawa, Hironori Washizaki and Yoshiaki Fukazawa
  The 6th IEEE International Conference on Software Testing, Verification and Validation, 9 pages, 2013. (accepted)
- • **POGen: A Test Code Generator Based on Template Variable Coverage in Gray-Box Integration Testing for Web Applications**
  Kazunori Sakamoto, Tomohiro Kaizu, Daigo Hamura, Hironori Washizaki and Yoshiaki Fukazawa
  The 16th International Conference on Fundamental Approaches to Software Engineering, pp.343-358, 2013. (accepted)
- ○ **A Framework for Analyzing and Transforming Source Code Supporting Multiple Programming Languages**
  Kazunori Sakamoto
  ACM Student Research Competition at the Aspect-Oriented Software Development 2013, 2 pages, 2013. (accepted)
- • **Estimate of the Appropriate Iteration Length in Agile Development by Conducting Simulation**
  Ryushi Shiohama, Hironori Washizaki, Shin Kuboaki, Kazunori Sakamoto, Yoshiaki Fukazawa
  Agile2012 Conference, pp.41-50, 2012.
- • **Towards a Unified Source Code Measurement Framework Supporting Multiple Programming Languages**
  Reisha Humaria, Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  The 24th International Conference on Software Engineering and Knowledge Engineering, pp.480-485, 2012.
- • **UniAspect: A Languagae-Independent Aspect-Oriented Programming Framework**
  Akira Ohashi, Kazunori Sakamoto, Tomoyuki Kamiya, Reisha Humaria, Satoshi Arai, Hironori Washizaki and Yoshiaki Fukazawa
  The 2nd Workshop on Modularity in Systems Software co-located with Aspect-Oriented Software Development 2013, pp.39-44, 2012.

- **Evaluation of Understandability of UML Class Diagrams by Using Word Similarity**
  Yuto Nakamura, Kazunori Sakamoto, Kiyohisa Inoue, Hironori Washizaki and Yoshiaki Fukazawa
  The Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, pp.178-187, 2011.

○ **A Tool For Detecting Duplicated Test Code Based On Test Coverage to Assist TDD**
  Kazunori Sakamoto, Takuto Wada, Hironori Washizaki and Yoshiaki Fukazawa
  Joint Workshop on Software Science and Engineering, pp.41-46, 2011.

○ **Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages**
  Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  The 10th International Conference on Quality Software, pp.262-269, 2010.

Domestic Conference Presentations

○ **Gray-box Integration Testing and Template Variable Coverage for Web Applications**
  Kazunori Sakamoto, Tomohiro Kaizu, Daigo Hamura, Hironori Washizaki and Yoshiaki Fukazawa
  JSSST Workshop on the Foundation of Software Engineering (ソフトウェア工学の基礎ワークショップ), pp.131-140, 2012. (in Japanese)

- **DePoT: Testing Framework for Web Application Test**
  Shohei Aoi, Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  JSSST Workshop on the Foundation of Software Engineering (ソフトウェア工学の基礎ワークショップ), pp.121-130, 2012. (in Japanese)

○ **UNICOEN: A Unified Framework for Code Engineering Supporting Multiple Programming Languages**
  Kazunori Sakamoto, Akira Ohashi, Daichi Ota, Hironori Washizaki and Yoshiaki Fukazawa

IPSJ/SIGSE Software Engineering Symposium (ソフトウェアエンジニアリングシンポジウム), pp.1-8, 2012. (in Japanese)

- **Topics of the Session on Software Design, Software Patterns and Agile Development in Winter Workshop 2012**
  Kazunori Sakamoto, Masanari Motohashi
  IPSJ Winter Workshop in Biwako (ウィンターワークショップ・イン・琵琶湖), pp.1-2, 2012. (in Japanese)
- **A Unified Source Code Measurement Tool Supporting Multiple Programming Languages**
  Reisha Humaria, Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  IPSJ Winter Workshop in Biwako (ウィンターワークショップ・イン・琵琶湖), pp.5-6, 2012.
- **Extenable Refactoring Engine for Multi Programming Languages**
  Tomoyuki Kamiya, Kazunori Sakamoto, Akira Ohashi, Hironori Washizaki and Yoshiaki Fukazawa
  IPSJ Winter Workshop in Biwako (ウィンターワークショップ・イン・琵琶湖), pp.23-24, 2012. (in Japanese)

○ **A Pattern Language for Programming Contest through Fight between Computer Players**
  Kazunori Sakamoto, Akira Ohashi, Masaya Shimizu, Syuhei Takahashi, Shinichi Murakami, Hironori Washizaki and Yoshiaki Fukazawa
  The 2nd Asian Conference on Pattern Languages of Programs, pp.1-18, 2011. (in Japanese)

- **A Technique for Generating UI Specifications and Test Specifications from UI Mock-ups for Web Applications**
  Kazunori Sakamoto, Masaharu Toukai, Yuko Murakami, Rie Miyahar, Yukiko Okumura, Koichi Akiyama, Hironori Washizaki, Yoshiaki Fukazawa
  Symposium on Software Quality Profession (ソフトウェア品質シンポジウム), pp.1-8, 2011. (in Japanese)
- **Estimating the appropriate iteration term of agile development using simulation**
  Ryushi Shiohama, Kazunori Sakamoto, Shin Kuboaki, Hironori Washizaki and Yoshiaki Fukazawa

IPSJ/SIGSE Software Engineering Symposium (ソフトウェアエンジニアリングシンポジウム), pp.1-6, 2011. (in Japanese)

- **Problem and Consideration in Testing Framework**
  Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  JSSST Workshop on the Foundation of Software Engineering (ソフトウェア工学の基礎ワークショップ), pp.193-194, 2010. (in Japanese)

○ **A Tool Detecting Redundant Test Code Based On Test Coverage**
  Kazunori Sakamoto, Takuto Wada, Hironori Washizaki and Yoshiaki Fukazawa
  IPSJ/SIGSE Software Engineering Symposium (ソフトウェアエンジニアリングシンポジウム), pp.133-138, 2010. (in Japanese)

Annual Convention Presentation

- **Suggestion of New Context-Oriented Programming Language Through Comparative Experiments with Object and Aspect-Oriented Programming Languages**
  Fumiya Kato, Kazunori Sakamoto, Hironori Washizaki and Yosiaki Fukazawa
  The 75th National Convention of IPSJ (情報処理学会 第 75 回全国大会), 2 pages, 2013. (submitted, in Japanese)
- **Suggestion of contribution-based gamified testing tool for education**
  Ryohei Takasawa, Kazunori Sakamoto, Hironori Washizaki and Yosiaki Fukazawa
  The 75th National Convention of IPSJ (情報処理学会 第 75 回全国大会), 2 pages, 2013. (submitted, in Japanese)
- **Construction of the platform for programming study for female students**
  Koichi Takano, Kazunori Sakamoto, Hironori Washizaki and Yosiaki Fukazawa
  The 75th National Convention of IPSJ (情報処理学会 第 75 回全国大会), 2 pages, 2013. (submitted, in Japanese)
- **Facilitate defect removal based on bug pattern using Gamification**
  Satoshi Arai, Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  The 75th National Convention of IPSJ (情報処理学会 第 75 回全国大会), 2 pages, 2013. (submitted, in Japanese)
- **Report on Winter Workshop 2012 in Biwako**
  Katsuhisa Maruyama, Takayuki Omori, Hiroshi Igaki, Masahide Nakamura,

Kyohei Fushida, Masateru Tsunoda, Hiroshi Kazato, Joji Okada, Kozo Okano, Kazunori Sakamoto, Masanari Motohashi, Tomoji Kishi, Natsuko Noda, Takashi Kobayashi and Shinpei Hayashi
The 178th Workshop of IPSJ Special Interest Group on Software Engineering (情報処理学会 第178回ソフトウェア工学研究発表会), pp.1-8, 2012. (in Japanese)

- **A Proposal of Gray-box Integration Testing for Web Applications**
  Kazunori Sakamoto, Tomohiro Kaizu, Daigo Hamura, Hironori Washizaki and Yoshiaki Fukazawa
  The 176th Workshop of IPSJ Special Interest Group on Software Engineering (情報処理学会 第176回ソフトウェア工学研究発表会), pp.1-8, 2012. (in Japanese)

○ **UNICOEN: A Unified Framework For Code Engineering Supporting Multiple Programming Languages**
Kazunori Sakamoto, Akira Ohashi, Daichi Ota, Hironori Washizaki and Yoshiaki Fukazawa
The 88th Workshop of IPSJ Special Interest Group on Programming (情報処理学会 第88回プログラミング研究発表会), pp.46, 2012. (in Japanese)

- **A Language Independent Aspect-Oriented Programming Framework**
  Akira Ohashi, Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
  The 28th Conference of JSSST (日本ソフトウェア科学会 第28回大会), pp.1-12, 2011. (in Japanese)

○ **A Tool for Measureing Test Coverage Metric for Programming Languages with Dynamic Evaluation**
Kazunori Sakamoto, Hironori Washizaki and Yoshiaki Fukazawa
The 27th Conference of JSSST (日本ソフトウェア科学会 第27回大会), pp.153-158, 2010. (in Japanese)

○ **Software Patterns for Game in Which Users Participate through AI Programming**
Kazunori Sakamoto, Akira Ohashi, Masaya Shimizu, Syuhei Takahashi, Shinichi Murakami, Satoru Uchiyama, Yuki Shiroma, Yutaro Nomoto, Akihiko Syoyama, Yuto Nakamura, Hironori Washizaki and Yoshiaki Fukazawa
Workshop on Software Pattern, Architecture and Agile Development (ソフト

ウェアのパターンとアーキテクチャ・アジャイル開発ワークショップ), pp.1, 2010. (in Japanese)

Other Publications and Presentations

- ○ **A Unit Testing Tool Supporting Multiple Programming Languages and Emerging Trends on Test Coverage**
  Kazunori Sakamoto
  GAIO Private Seminar 2012 Fall (ガイオ プライベートセミナー 2012 秋), 2012. (in Japanese)
- • **Principles and Patterns on Game AI Programming Contests such as Samurai Coding**
  Hironori Washizaki, Kazunori Sakamoto, Masahiko Wada
  CESA Developer Conference (CESA デベロッパーカンファレンス), 2012. (in Japanese)
- ○ **UNICOEN: A Unified Framework for Code Engineering Supporting Multiple Programming Languages**
  Kazunori Sakamoto, Hironori Washizaki
  The 2nd NII-PKU International Joint Workshop on Advanced Software Engineering, 2011.
- • **A Software Architecture Suitable for Educational Game System of AI Programming**
  Kazunori Sakamoto
  CESA Developer Conference (CESA デベロッパーカンファレンス), 2010. (in Japanese)