# IUT Design Document

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Description** | **Date** | **Author** |
| 1.0 | Initial version |  | Weizhao |
| … |  |  |  |
| 1.2 | Add implementation structure overview, refine document structure | 2016-5-6 | Ziyi |
| 1.2.1 | Refine 3.1.4 by adding disadvantages of AspectC++ | 2016-5-10 | Weizhao |
| 1.2.2 | Add 3.3 Case Stuty: CCTZ  Add 5 Gtest | 2016-5-15 | Feiyue |
| 1.2.3 | Modify 3.3 Case Study | 2016-5-25 | Feiyue |
| 1.2.4 | Modify 6 Benchmark and Add 7 Milestone | 2016-5-26 | Feiyue |
| 1.2.5 | Add 4 Differ | 2016-6-26 | Weizhao |

Index

[IUT Design Document 1](#_Toc457778599)

[Index 1](#_Toc457778600)

[1 Problem Description 3](#_Toc457778601)

[2 Implementation Overview 3](#_Toc457778602)

[3 Test-Function Mapper 3](#_Toc457778603)

[3.1 AspectC++ (Aspect-Oriented Programming) 3](#_Toc457778604)

[3.1.1 Basic Implementation 4](#_Toc457778605)

[3.1.2 Pros and Cons 4](#_Toc457778606)

[3.2 gcov 6](#_Toc457778607)

[3.2.1 Basic Implementation 6](#_Toc457778608)

[3.2.2 Pros and Cons 7](#_Toc457778609)

[4 Differ 8](#_Toc457778610)

[4.1 Tools 8](#_Toc457778611)

[4.1.1 Clang AST 8](#_Toc457778612)

[4.1.2 Eclipse CDT 9](#_Toc457778613)

[4.2 Strategy 9](#_Toc457778614)

[4.2.1 File Differ 9](#_Toc457778615)

[4.2.2 AST Differ 9](#_Toc457778616)

[5 Gtest 11](#_Toc457778617)

[6 Benchmark 11](#_Toc457778618)

[7 Milestones of IUT 12](#_Toc457778619)

[8 Limitation 13](#_Toc457778620)

[9 User Guide 13](#_Toc457778621)

# Problem Description

In order to figure out the mapping relationship between the test cases and the functions in the source code of the target project, an approach is needed to statically or dynamically trace the execution of every function (including inline function or virtual function etc.) and record it.

# Implementation Overview

Project

v1

Test-Function Mapper

Project

v2

DB

DB Operator

Differ

Tests

Affected Tests

Executor

GCov

AspectC++

CDT

Clang

# Test-Function Mapper

## AspectC++ (Aspect-Oriented Programming)

Aspect-oriented programming (AOP) allows modularizing cross-cutting concerns in a single module, an aspect. Aspects can modify existing classes, but most commonly they provide 'advice' that runs before, after, or around existing functionality.

AspectC++ is an aspect-oriented extension of C and C++ languages.

For the IUT project, AspectC++ enables us to log the “signature” (type, name and scope) of every function before, after, or around its execution WITHOUT revising the source code of the target project.

### Basic Implementation

#### Coding

There are several concepts in AspectC++ which are essential for writing AspectC++ codes such as *aspect, advice, join* *point and pointcut etc.* (refer to http://www.aspectc.org/doc/ac-quickref.pdf).

Aspects are a special AspectC++ language element, which can be used to implement crosscutting concerns in separate modules. Aspect definitions have to be implemented in special “aspect header files”, which normally have the filename extension “.ah”. And in the definition of an *aspect*, we can define *pointcut* to describe where we want our aspect code to be “woven” and define *advice*s that execute aspect code when the target program reaches the *join point*s of the *pointcut*.

#### Compiling (Weaving)

AspectC++ requires an specific compiler to “weave” the code we write in the “.ah” file to the target program – ac++/ag++.

The program ac++ is a compiler for the AspectC++ programming language. It is implemented as a preprocessor that transforms AspectC++ code into ordinary C++ code. After the code transformation the output of ac++ can be compiled to executable code with ordinary C++ compilers like GNU g++.

The ag++ program provides a more intuitive frontend to ac++ in a GNU environment. It basically wraps the functionality of the aspect weaver and the c++ compiler into one single program.

The usage of ac++/ag++ is similar to g++, a simple example (assume “.ah” files share the same directory of the “main.cc” file) is

“ag++ main.cc –o main”

### Pros and Cons

#### Pros

1. Based on AspectC++ APIs, simple to implement.

This method is based on AspectC++, a well-built extension for C/C++, which provides rich APIs for us to use to solve our problem. The idea of AOP has been popular for years and has been widely known and accepted. It’s very simple for us to follow the AspectC++ reference to write aspect code and compile it.

1. No change of source code the target project.

AspectC++ allows us to weave our aspect code into the target program without modifying its code. It simply uses *advice* which runs before, after, or around existing functionality. Therefore, there is no need to make an extra copy of the target project and modify it.

1. Few LoC, reduce the possibility of error.

With appropriate uses of AspectC++ APIs, it’s easy for us to achieve our goal with few lines of code. The *match expression*s provided by AspectC++ allows us to easily match certain functions or variables with a given pattern. With less code comes less possibility of error of the program we write, which enhances robustness of the program.

#### Cons

1. Require a specific compiler to compile.

As is mentioned above, AspectC++ needs specific compiler to compile the aspect code, weaving it to the target program. Nevertheless, the usage of the specific compiler is quite similar to GNU g++. Actually, ag++ basically wraps the functionality of the aspect weaver and the c++ compiler into one single program.

1. As for ac++, it’s practically not feasible use ac++ to weave aspects into target code. Because the path of every file that is included in the target code must be specified using the “-I <path>” flag of ac++ command, including even the standard libraries such as “iostream”. As it’s quite difficult and complex to include all the paths, ac++ cannot satisfy our requirement.
2. As for ag++, it’s possible to using ag++ to do the weaving job. We first need to using a C++ compiler (e.g. GNU g++, Clang c++) to generate dependency files, and then use ag++ to weave the code and generate the object files, and finally we can use the C++ compiler to link the object files to generate executables. It easy when the project structure is very simple and it’s compiled by typing commands in terminal, but when there is a big project with many source files and is built by makefile file, it would be quite tricky to adopt ag++ to the project.
3. Need to learn AspectC++.

It takes time to get familiar with the AspectC++ language elements and their usage. And There isn’t much support to find online for asking about issues that we may encounter during AspectC++ development.

1. Might have performance problem.

We’ve applied AspectC++ on some very simple C++ programs, add compared the time cost of compilation between ag++ and g++. It can be known from the result that the “weaving” process can cost some overhead. However, the time of compilation using ag++ seems to be nearly the same. The factors that affect the performance cannot be sure until we try some larger projects with more source code and more files.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Name* | #source file | #aspect file | Time for g++  (s) | Time for ag++  (s) |
| Helloworld | 2 | 1 | 0.267 | 1.210 |
| Modules | 2 | 4 | 0.030 | 1.173 |
| Coverage | 1 | 2 | 0.030 | 1.894 |
| Profiling | 1 | 2 | 0.032 | 1.204 |
| Singleton | 3 | 2 | 0.034 | 1.173 |
| Another\_demo | 5 | 1 | 0.272 | 1.188 |

1. AspectC++’s support for macros is poor.

In the manual for ac++ (<http://aspectc.org/doc/ac-compilerman.pdf)>, “Macros” is marked as “Unimplemented Features”.

1. AspectC++’s support for filename extensions is disappointing.

By default, AspectC++ only support source files with “.cc” extension. Although a “-e <extension>” flag is to specify the file extension name, it’s still tricky to use this flag.

## gcov

gcov is a code coverage tool for C or C++. Add CXXFLAGS = -fprofile-arcs -ftest-coverage to makefile and add -lgcov when link. Run the target program. gcov -b <filename>. <filename>. gcov is a detailed coverage report file which contains function coverage information.

The goal is to obtain, for each function, whether it is entered. gcov is a code coverage tool for C/C++, which can be used here.

### Basic Implementation

#### Modify makefile

Add -fprofile-arcs -ftest-coverage to CXXFLAGS. Add -lgcov when linking.

#### Make and run

Make the project and run it. You will get .gcno and .gcna file. Do not delete them.

#### gcov -b filename

gcov [options] files

gcov accepts many options. We use -b (short for –branch-probabilities). Write branch frequencies to the output file, and write branch summary info to the standard output. This option allows you to see how often each branch in your program was taken. Unconditional branches will not be shown, unless the -u option is given. The function level summaries are shown when using -b. Traverse the directory recursively and invoke gcov. Some .gcov files are generated after invoking gcov. They are human-readable text files which describe the coverage information.

#### Analyse \*.gcov

Analyse all .gocv files. The .gocv file list the source code appending some coverage information. The function level information is formatted as “function \_Z6globalv called 5 returned 100% blocks executed 100%”. Identify the pattern and collect the relevant data – whether a function is executed or not.

### Pros and Cons

#### Pros

It does not need too much extra code. A basic implementation needs less then 200 LOC.

#### Cons

It does more work than we need because it counts execution times of every line. This will waste some time.

It is not easy to modify makefile for large-scale project.

**3.3 Case Study: CCTZ**

CCTZ is a C++ library for translating between absolute and civil times using the rules of a time zone. It has 6603 lines of code.

The testing is build by bazel by default in this project. We prefer makefile. Install gtest to system(http://stackoverflow.com/questions/13513905/how-to-setup-googletest-as-a-shared-library-on-linux). Modify makefile to build the testing object file. Our program can successfully get the coverage information of every function in the project. But gcov also records the libraries' execution like stl and gtest which are useless. We only analyse the code of project by checking the filename.

Steps:

1. List all test cases;

2. Run each test case and get coverage.

It costs about 15 seconds to get function coverage of 57 test cases.

Problems:

1. Functions at runtime cannot map to functions in source code such as template, default constructor.

2. Function naming. Gcov has its convention which we have to map them to ast version.

# Differ

When a new version of the target project arrives, in order to test it incrementally, we first to do a “diff” between the old version and the new version. The “diff” process can be divided into two levels, file level and AST (Abstract Syntax Tree) level. First we compare the two versions of source code in the file level to see if any file or folder has been added, deleted or modified. When a file is detected to be modified, we then compare the old file and the new file in the AST level, that is we transform the source code into AST and compare different kinds of AST nodes on it, and extract the exact information about which function (and non-function element) has been changed.

## Tools

The tools to support Differ are Clang AST and Eclipse CDT (C/C++ Development Tooling). And the final decision is using Eclipse CDT.

### Clang AST

Clang(<http://clang.llvm.org/docs/index.html> ) is a compiler front end for the programming languages C, C++, Objective-C, Objective-C++ etc. And Clang AST is targeted at developers who either want to contribute to Clang, or use tools that work based on Clang’s AST, like the AST matchers.

Clang provides three different levels of tooling, which are LibClang, Clang Plugins and LibTooling. Each level provides different ways to deal with AST(<http://clang.llvm.org/docs/Tooling.html> ).

Clang’s AST nodes are modeled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types like Decl and Stmt. Many important AST nodes derive from Type, Decl, DeclContext or Stmt, with some classes deriving from both Decl and DeclContext.

There are also a multitude of nodes in the AST that are not part of a larger hierarchy, and are only reachable from specific other nodes, like CXXBaseSpecifier.

Thus, to traverse the full AST, one starts from the TranslationUnitDecl and then recursively traverses everything that can be reached from that node - this information has to be encoded for each specific node type. This algorithm is encoded in the RecursiveASTVisitor. See the RecursiveASTVisitor tutorial.

The two most basic nodes in the Clang AST are statements (Stmt) and declarations (Decl). Note that expressions (Expr) are also statements in Clang’s AST.

Clang has a built-in AST-dump mode, which can be enabled with the flag “-ast-dump”.

### Eclipse CDT

Eclipse CDT(<http://www.eclipse.org/cdt/> )is a C/C++ development tool plugin for Eclipse. It provides resourceful API for parsing C/C++ code and manipulate AST.

A C/C++ source file can be parsed into an instance of IASTTranslationUnit which represents an AST of the source file. This IASTTranslationUnit contains all the information we want; we just have to figure out what to do with it.

## Strategy

As is mentioned above, the “diff” process can be divided into two levels, file level and AST (Abstract Syntax Tree) level. First we compare the two versions of source code in the file level to see if any file or folder has been added, deleted or modified. When a file is detected to be modified, we then compare the old file and the new file in the AST level, that is we transform the source code into AST and compare different kinds of AST nodes on it, and extract the exact information about which function (and non-function element) has been changed.

### File Differ

The File Differ is to compare two versions of source code in the level of directories and files to see whether any file has been added, deleted or modified.

The diff strategy here is to use a Map (HashMap) to store every key-value pair of the new folder where the key is file name and the value is an instance of java.io.File. And we traverse the files in the old folder to see if the file names in the old folder can be found in the Map.

If we can’t find a match in the Map, then this file must be deleted from the old version.

If we can find a match, then we have to see if the contents of the two versions are identical. If they are the same, we then remove the entry from the Map. If not, we then use the AST Differ to find the differences between the two files, and also remove the entry from the Map.

Finally, we look into the Map if there’s anything left after removing the matched files. The ones that have been left are all files that are added in the new version.

### AST Differ

The diff strategy for AST Differ is similar to that for File Differ; the only difference is to change the value of the Map entry to whatever kind of AST node we want to diff and the key to the corresponding string that can identify the node. Now let’s see how to find different kinds of AST node in an AST.

Following the visitor design pattern, the Eclipse CDT API provides a class IASTVisitor to visit different nodes of an instance of IASTTranslationUnit. User can implement their own visitor by extending IASTVisitor, and override the visit methods whose parameters are different kinds of nodes.

#### Functions

In the Eclipse CDT API, functions in C/C++ code are all identified as an instance of IASTFunctionDefinition. Using the AST visitor, we can easily get all the functions in an AST.

#### Non-function nodes

For non-function nodes, there are two ways to capture them.

The first is to use the ASTGenericVisitor which enables only one visit methods whose parameter is an instance of IASTNode which is the super class of all AST nodes. When we find that the node being visited is not a function, we then know that it’s a non-function node, and we can go on to handle it. In this way all the non-function node can be captured.

The second way is to identify specific nodes using IASTVisitor. Currently we can identify several cases as follows.

|  |  |  |
| --- | --- | --- |
| Case | Description | example |
| Macro Definition | Macro defined in the source files of the target project, can be divided into Object style and Function style. | #define PI 3.14 //object  #define P(x) x //function |
| Class Definition | Class defined in the source file. | Class A{}; |
| Namespace Definition | Namespace defined in the source file. |  |
| Other Simple Definition | Common definitions such as global variable, class field members etc.  We can now detect whether a simple definition is a field member definition. | int a = 0;  ClassFoo foo; |
| Problem | Code that cannot be recognized, such as macro whose definition cannot be found, and common syntax errors. | TEST(…){} //as we encountered in gtest  int b //missing “;”  int //meaningless |

# Gtest

This sections gives a basic introduction of gtest which is necessary for the iut project. Details can be found in the official website.

A gtest file is a .cpp file which has many tests. A standalone executable object file is generated after compilation. You can run it and it will show you the results of tests. Some options are helpful:

--help

Print the help

--gtest\_list\_tests

List the names of all tests instead of running them. The name of TEST(Foo, Bar) is "Foo.Bar".

--gtest\_filter=POSTIVE\_PATTERNS[-NEGATIVE\_PATTERNS]

Run only the tests whose name matches one of the positive patterns but none of the negative patterns. '?' matches any single character; '\*' matches any substring; ':' separates two patterns.

# Benchmark

|  |  |  |
| --- | --- | --- |
| Project | LOC | Link |
| cctz | 6603 | https://github.com/google/cctz |

# Milestones of IUT

|  |  |  |
| --- | --- | --- |
| **Date (2016)** | **Java** | **C++** |
| **4.13** |  | An instrumentation program that enable recording function calls. |
| **5.4** | Java IUT with a simplified workflow | A diff program that finds difference between two versions of the target project. |
| **5.25** | Java IUT with a simplified and optimized workflow | Continue to implement and compare AspectC++ and gcov methods. Choose one. Continue to finish 'diff'. Learn to run certain test cases under gtest framework, mapping functions to test cases. |
| **6.15** |  | Select test cases based on function coverage and program differences. Set up database, utility for database operations. A basic runnable version of iut for java.  C++ workflow ready (function-testcase mapping, diff, and select and run test case) |
| **7.6** |  | Support git commit integration.  Simplify the workflow |
| **7.25** | IUT with both Java and C++. Provide unified interface. | IUT with both Java and C++. Provide unified interface.  The entire simplified workflow is ready,  clear documentation about the design decisions, and limitations, and 2 demos, 1 for java, one for C++, could be smoothly run to demo the workflow. |

Specific plans since 6.15:

Week 1: Improve usability. User can choose all-in-one command or subcommand. Function coverage persistency support.

Week 2: Identify differences out of functions.

Week 3: Deal with special cases like macro.

Week 4: Simpify workflow of JavaIUT.

Week 5: Conduct a case study. Record a demo.

Week 6: Revise documentation and user manual.

# Limitation

1. Test runs will generate too many logs when instrumenting a log statament in the beginning of each functions without removing duplicates. For example, cctz\_v1\_test in cctz prints 500 0000+ logs. This cost much time. A possible solution is removing duplicates in c++ code which is inserted.

2. Unable to deal with some macro which is a user-defined grammer and cannot be recognized without expansion.

3. Modifications out of functions are not precisely handled.

4. We have not tested the tool over a wide range of projects and only target on project cctz now. So it possibly needs some manual effort to accommodate it to more projects.

# User Guide