Challenges in Automated Unit Testing of Complex C++ Programs

Letian Zhang[O] 이아청 김문주

KAIST KAIST KAIST

zhangletian630@gmail.com ahcheong.lee@kaist.ac.kr moonzoo.kim@gmail.com

Abstract

Clementine is an automated C++ unit-testing tool to analyze source code and generate random method call sequences. The key challenge of generating unit tests for real-world C++ programs is the severe diversity of C++ syntax features. Because of the diversity, Clementine suffers huge amounts of uncompilable test cases. The two objectives of this research are 1) to prevent Clementine from generating uncompilable test cases and 2) to support Clementine to generate at least one compilable test case for a broader range of functions.

In this paper, we enumerate the examples of complex C++ syntaxes that resulted in test case generation failure. By addressing those problems, we could improve Clementine's reliability and usability for developers. We applied the fixed Clementine on 16 real-world C++ programs, and the results show that 1) Clementine could reduce 94.6% uncompilable test codes, and 2) it could generate at least one compilable test code for all (100%) target functions in the programs.

1. Introduction

Unit testing in C++ presents unique challenges owing to certain features inherent in the language. C++ is renowned for its complexity, extensive feature set, and close-to-the-hardware capabilities, making it a powerful but intricate language to work with [1]. This complexity often translates into difficulties in designing and executing effective unit tests. There are some examples of automated unit-level testing tools for other programming languages (Randoop [2] and EvoSuite [3] for Java programs Pynguin [4] for Python programs.) Unfortunately, there are only a few automated unit-level tests for C++ programs due to complex C++ features. The intricacies of manual memory management, intricate pointer arithmetic, and the potential for undefined behavior make it challenging to ensure the correctness and reliability of C++ code. Despite these challenges, the importance of unit testing in the C++ development process cannot be overstated. Robust unit tests serve as a fundamental tool for identifying bugs early in the development lifecycle, enhancing code maintainability, and supporting the overall stability and longevity of C++ projects. In this context, addressing the challenges associated with C++ unit testing becomes imperative for ensuring the delivery of high-quality, reliable software. Additionally, a recent study reported that automated software testing produces test cases with higher test coverage compared to manually written test cases [5]

Clementine is a continuation of CITRUS [1]. It is an automated C++ unit-testing tool designed to analyze source code and generate method call sequences. In contrast to earlier automated

This research was supported by the National Research Foundation grants funded by the Korea government (NRF-2021R1A5A1021944, NRF-2021R1A2C2009384).

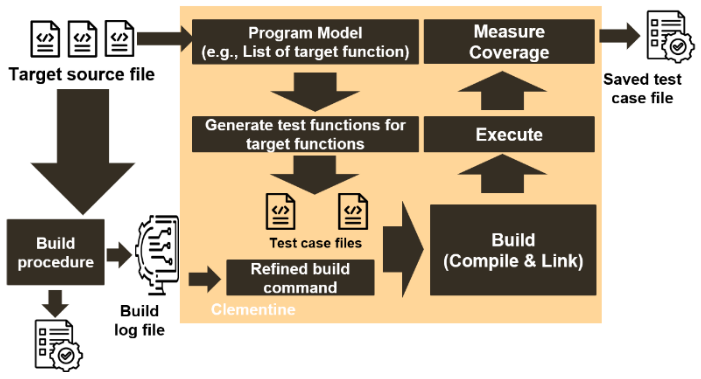


Figure 1 Clementine process

test tools designed for C++, Clementine stands out by offering support for critical features such as template processing and STL classes. Notably, it surpasses competitors like KLOVER, FSX, and UTBotCPP, which lack this extended functionality. Additionally, Clementine uniquely addresses non-public member functions, a capability absents in KLOVER, FSX, CITRUS, and UTBotCPP. We applied Clementine on 16 real-world projects (shown in Table 1) and found that Clementine generates a lot of uncompilable test code, and also it fails to generate test code for many functions under test. So, we made improvements and refinements to address these two aspects, and we eventually reduced the number of uncompilable test functions generated from 4091 to 221 (reduced 94.6%), and the number of failures where Clementine failed to generate test functions was reduced from 6567 to 0 (100%).

```cpp
struct StructWithDefaultConstructor {
  int data;
  // Default constructor
  StructWithDefaultConstructor() : data(0) {}
};
struct StructWithoutDefaultConstructor {
  int data;
  // Parameterized constructor
  StructWithoutDefaultConstructor(int value) : data(value) {}
};
void test_function() {
  // compilable
  StructWithDefaultConstructor array_1[5];
  // uncompilable, since there is no default constructor
  StructWithoutDefaultConstructor array_2[5];
}
```

Figure 2 struct examples

2. Diverse problematic C++ syntax features

In this section, we explore a spectrum of challenges posed by various aspects of C++ syntax within the framework of Clementine's testing methodology. The intricacies can be broadly categorized into distinct themes, each demanding unique consideration.

2.1 Specialized Classes

When utilizing the object factory to generate product objects, which are represented as pointers, the dereferencing of these pointers takes place during the invocation of specific functions or when passing the object as an argument to the target function. However, there are numerous reasons in C++ why the act of dereferencing a pointer may be deemed illegal. These reasons encompass a wide range of scenarios, and some of them include:

a. The target pointer points to an abstract class, no direct instances of it can be created [7]

b. The copy constructor of the class pointed to by the target pointer is not accessible (non-public access, deleted).

To solve these problems, Clementine construct an inheritance tree by analyzing the source code and following the inheritance tree to find the subclasses of the class that do not have these characteristics.

2.2 Array Creation

Clementine encounters difficulties when attempting to create an array of structs due to the absence of default constructors for some structs [8]. To address this, we analyzes the constructor requirements of various structures and implements a solution that involves calling the constructor repeatedly, ensuring successful array creation regardless of the struct's characteristics.

2.3 Target Type Extraction as clang:: RecordType

In C++, as an object-oriented programming language, the vast majority of function calls need to be realized by creating objects of classes. Clementine is a clang-based automated testing tool, so when creating test functions for these target functions, it happens from time to time that the target class is extracted as clang::RecordType. But the complexity of clang AST can lead to a variety of scenarios where it doesn't succeed in getting the target class as the clang::RecordType we need, for example:

a. Injected-class-name (Figure 3)

b. Type alias by C++ keyword using [9]

```cpp
template <typename T>
class Option {
public:
  Option();
  Option& operator=(const Option<T>& other);
}
```

Figure 3 Injected-class-name example

This ultimately leads to Clementine failing to generate test functions for such a target function. To solve such a problem, get the underlying clang::RecordType by performing a desugar operation

2.4 Template Functions

Another notable feature of the C++ language is Template Functions [10]. The need for complex and large template functions has led Clementine to fail to generate test functions for such functions Since A template class in C++ has two primary components: template parameters and template specialization. I'll explain the problems we found by the difference in the two parts where they occurred:

a. Template specialization

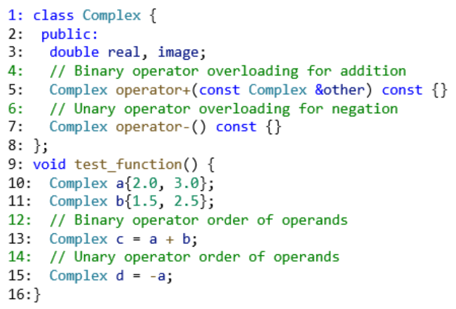
Among the problems we've found with Template Specialization is that the source code doesn't contain the specialization of the target template or contains the specialization but is invalid. The solution is that if no valid specialization is found during the analysis of the target source code, then create a type alias for the target template through the using statement, replacing the template type parameter with a simple built-in type.

b. Template Parameters

One example of this problem is that when Clementine targets a non-template member class of a template class, it treats the class as a separate template class but fails to get the template parameter of that member class. To solve this problem, if the parent class of the target class is a template class and the target class itself is not a template class, then only the parent class is treated as a template class.

2.5 Operator Overloading Call Statement.

There are many overloaded operators in C++ classes (Figure 4), to test these overloaded operators, Clementine needs to correctly identify how the operator is called, i.e., identify whether it is the target class that is being invoked or a pointer to the target class. In addition, some special operators require special handling, such as the correct order of operands and how to write statements



```
class Complex {
 public:
  double real, image;
  // Binary operator overloading for addition
  Complex operator+(const Complex &other) const {}
  // Unary operator overloading for negation
  Complex operator-() const {}
};
void test_function() {
  Complex a{2.0, 3.0};
  Complex b{1.5, 2.5};
  // Binary operator order of operands
  Complex c = a + b;
  // Unary operator order of operands
  Complex d = -a;
}
```

Figure 4 overloaded operator example

that call the target overloaded operator with the correct syntax rules.

3. Experiment Evaluation

3.1 Experiment Setup

To be able to discover more fully the problems of Clementine in generating test functions for objective functions, we applied Clementine to 16 real-world subjects. In total, there are 63,442 functions to be tested.

3.2 Experiment Result

The number of uncompilable test functions generated is reduced from 4091 to 221 (94.6%), and the number of failures where Clementine failed to generate test functions is reduced from 6567 to 0 (100%).

Table 1. 16 real-world C++ open-source projects used for evaluation.

Name	Size (Loc)	Name	Size (Loc)
clip	17,000	hjson	2,911
Exiv2	83,011	JsonBox	1,477
gflags	3,954	jsoncpp	5,420
glog	9,510	json-voorhees	8,614
guetzli	8,029	javr	4,860
PcapPlusPlus	64,805	re2	20,373
sql-parser	13,332	tinyxml2	3,606
xpdf	125,529	yaml-cpp	8,800

4 Conclusion and Future Works

In conclusion, this paper addressed challenges in Clementine, a C++ automated unit-testing tool, specifically focusing on issues related to test case generation. Our research resulted in substantial improvements to Clementine's source code, reducing the number of unworkable test cases and successfully generating tests for previously problematic functions.

Despite Clementine's demonstrated capabilities, challenges persist in generating uncompilable test code, stemming from issues such as accessing non-public member functions or variables and handling template structs incorrectly. In future work, we plan to develop mechanisms for correct access those problems. These collaborative efforts, accompanied by thorough testing and evaluation, aim to establish a more robust and reliable testing framework within Clementine, advancing its capabilities in generating compilable test code.

References

[1] R. S. Herlim, Y. Kim, and M. Kim, CITRUS: Automated Unit Testing Tool for Real-world C++ Programs, IEEE International Conference on Software Testing, Verification and Validation (ICST) Testing Tools track, April 4 - 13, 2022

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-Directed Random Test Generation,” Proceedings of the 29th International Conference on Software Engineering, (Washington, DC, USA), pp. 75–84, IEEE Computer Society, 2007

[3] G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-oriented Software,” Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, (New York, NY, USA), pp. 416–419, ACM, 2011

[4] S. Lukasczyk, F. Kroiß, and G. Fraser, “Automated Unit Test Generation for Python,” in Proceedings of the 12th Symposium on Search-based Software Engineering (SSBSE 2020, Bari, Italy, October 7–8), vol. 12420 of Lecture Notes in Computer Science, pp. 9–24, Springer, 2020

[5] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study,” ACM Trans. Softw. Eng. Methodol., vol. 24, Sept. 2015.

[6] F.Celler, “Programming Style Guidelines: ArangoDB Edition” Version 1.2.0

[7] Sakkinen, Markku. Inheritance and other main principles of C++ and other object-oriented languages. No. 20. University of Jyväskylä, 1992.

[8] E. El-Qawasmeh and B. Mahafzah, "Investigation of C++ Constructor Fail Features," 2000.

[9] H. D. Pande and B. G. Ryder, "Static type determination and aliasing for C++," Rutgers University, 1990.

[10] Siek, Jeremy, and Walid Taha. "A semantic analysis of C++ templates." European Conference on Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.