

Focal Methods for C/C++ via LLVM: Steps Towards Faster Mutation Testing

Sten Vercammen¹, Serge Demeyer^{1,2} and Lars Van Roy¹

¹University of Antwerp, Middelheimlaan 1, 2020 Antwerp, Belgium

²Flanders Make, Oude Diestersebaan 133, 3920 Lommel, Belgium

Abstract

Mutation testing is the state-of-the-art technique for assessing the fault detection capacity of a test suite. Unfortunately, it is seldom applied in practice because it is computationally expensive. In this paper we explore the use of fine-grained traceability links at the method level (named *focal methods*), to drastically reduce the execution time of mutation testing, by only executing the tests relevant to each mutant. In previous work for Java programs we achieve drastic speedups, in the range of 530x and more. In this paper we lay the foundation for identifying such focal methods under test in C/C++ programs by relying on the LLVM compiler infrastructure. A preliminary investigation on an 3,5 KLOC C++ project illustrates that we can correctly identify the focal method under test for 47 out of 61 tests,

Keywords

Mutation testing, Focal methods, Traceability

1. Introduction

Software testing is the dominant method for quality assurance and control in software engineering [1, 2], established as a disciplined approach already in the late 1970's. Originally, software testing was defined as “*executing a program with the intent of finding an error*” [3]. In the last decade, however, the objective of software testing has shifted considerably with the advent of continuous integration [4]. Many software test cases are now fully automated, and serve as quality gates to safeguard against programming faults.

Test automation is a growing phenomenon in industry, but a fundamental question remains: How trustworthy are these automated test cases? Mutation testing is currently the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [5]. The technique systematically injects faults into the system under test and analyses how many of them are killed by the test suite. In the academic community, mutation testing is acknowledged as the most promising technique for automated assessment of the strength of a test suite [6]. One of the major impediments to industrial adoption of mutation testing is the computational costs involved: each individual mutant must be deployed and tested separately [5].

For the greatest chance of detecting a mutant, the entire test suite is executed for each and every mutant [7]. As this consumes enormous resources, several techniques to exclude test


BENEVOL'21: The 20th Belgium-Netherlands Software Evolution Workshop, December 07–08, 2021, 's-Hertogenbosch (virtual), NL

✉ Sten.Vercammen@uantwerpen.be (S. Vercammen); Serge.Demeyer@uantwerpen.be (S. Demeyer)

🆔 0000-0002-9140-1488 (S. Vercammen); 0000-0002-4463-2945 (S. Demeyer)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

cases from the test suite for an individual mutant have been proposed. First and foremost, test prioritisation reorders the tests cases to first execute the test with the highest chance to kill the mutants [8]. Second, program verification excludes test cases which cannot reach the mutant and/or which cannot infect the program state [9]. Third, (static) symbolic execution techniques identify whether a test case is capable of killing the mutant [10, 11]. This paper explores an alternative technique: fine-grained traceability links via *focal methods* [12].

By using focal methods, we can establish a traceability link at method level between production code and test code. This allows us to identify which test cases actually test which methods and vice versa, hence drastically reduce the scope of the mutation analysis to a fraction of the entire test suite. We refer to this as goal-oriented mutation testing, and argue that the approach could be used to selectively target the parts of a system where mutation testing would have the largest return on investment.

In previous work, we estimated on an open source project (Apache Ant) that such goal-oriented mutation testing allows for drastic speedups, in the range of 530x and more [13]. In this paper we lay the foundation for identifying such focal methods under test in C/C++ programs by relying on the LLVM intermediate code as emitted by the CLANG compiler infrastructure.

2. Focal Methods under Test

Method invocations within a test case play different roles in the test. A majority of them are ancillary to a few ones that are intended to be the actual (or focal) methods under test. More particularly, unit test cases are commonly structured in three logical parts: setup, execution, and oracle. The setup part instantiates the class under test, and includes any dependencies on other objects that the unit under test will use. This part contains method invocations that bring the object under test into a state required for testing. The execution part stimulates the object under test via a method invocation, i.e., the *focal method* of the test case [12, 14]. This action is then checked with a series of inspector and assert statements in the oracle part that controls the side-effects of the focal invocation to determine whether the expected outcome is obtained.

Algorithm 1 represents a unit test case of a savings account where the intent is to test the *withdraw* method. For this, an account to test the *withdraw* method must exist. Thus, an account is created on line 3 (in Algorithm 1). To deposit or withdraw money to/from an account, the user must first authenticate himself (line 4). To make sure that the account has money to withdraw, a deposit must be made (line 5). If the savings account has a sufficient amount of money, the *withdraw* method will withdraw the money from the account (line 7), and the remaining amount of money in the savings account should be reduced. The latter is verified using the *assert* statement on line 11.

In the example, the intent clearly is to test the *withdraw* method. This method is the focal method as it is the last method that updates the internal state of the *account* object. The expected change is then evaluated in the oracle part by observing the result of the focal method (line 10), as well with the help of the *getBalance* method which only inspects the current balance.

Therefore, focal methods represent the core of a test scenario inside a unit test case. Their main purpose is to affect an object's state that is then checked by other inspector methods whose purpose is ancillary.

Algorithm 1 Exemplary Unit Test Case for Money Withdrawal

```
1: function TESTWITHDRAW
2:   ▷ Setup: setup environment for testing
3:   account ← CREATEACCOUNT(account, auth)
4:   account.AUTHENTICATE(auth)
5:   account.DEPOSIT(10)
6:   ▷ Execution: execute the focal method
7:   success ← account.WITHDRAW(6)
8:   ▷ Oracle: verify results of the method
9:   balance ← account.GETBALANCE()
10:  ASSERTTRUE(success)
11:  ASSERTEQUAL(balance, 4)
```

2.1. Limiting Test Scope for Mutation Testing

To adopt the focal method under test heuristic for mutation testing, we assume that a given test method does not suffer from the *eager test* code smell [15]. This is good practice anyway as it increases the diagnosability of the test cases. When this assumption holds, a test method *testF* is specifically designed to test a method *f* and not a series of other methods (*a*, *b*, and *c*). If method *f* would be faulty (i.e. include a mutant), then *testF* is responsible to detect this. If *f* calls methods *a*, *b*, and *c*, then *testA*, *testB*, and *testC* are responsible for detecting faults in their respective methods. Therefore, *testF* is not required to detect a fault if the fault is inside method *a*, *b*, or *c*. We thus argue that given a faulty method *f*, it should suffice to only execute those test cases which are responsible for testing the method *f*.

Under the premise that it is the responsibility of the (few) test cases that test a focal method *f* to catch all faults in the method *f*, we can assume that it suffices to limit the test scope to these selected test cases when we are killing mutants in method *f*. We assume even if we exclude those test cases that only happen to call a method *f* in one of its routines, there ought to be a simpler test case which tests the method *f* as a focal method that ought to also reveal that the method is faulty as that test case bears the responsibility to test the method and not the more complex test case.

Applied to mutation testing, this means that if a mutant is located in method *f*, we only need to execute these (few) test cases that test the focal method under test *f*.

3. The LLVM Compiler Infrastructure

The original tool for identifying the focal method under test was developed for java programs [12, 14]. No such tool exists for C/C++ hence we set out to investigate whether we can adopt the LLVM Compiler Infrastructure to expose the test-to-method relationship. LLVM is a set of compiler and toolchain technologies which is generally used to provide an intermediary step in a complete compiler environment. The idea is that any high level language can be converted to an intermediary language. This intermediary language will then be further optimised using an aggressive multi-stage optimisation provided within LLVM to then be converted to machine-

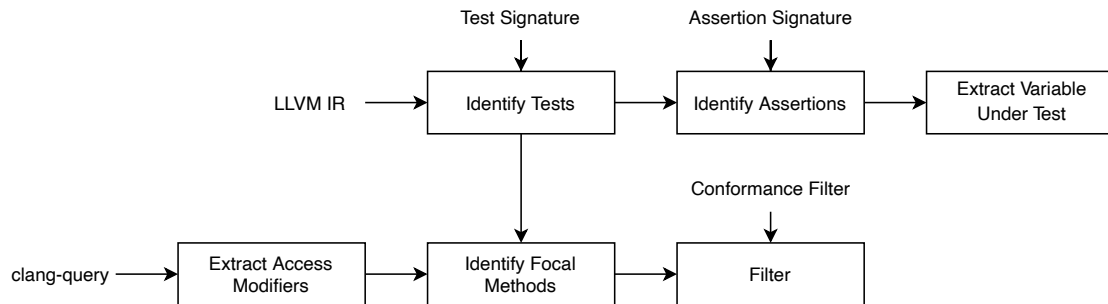
dependent assembly language code [16].

Specifically, for this research we focus on the intermediary language called LLVM IR, where IR stands for Intermediary Representation. This is a low-level programming language, similar to assembly, that is easily understandable and readable. Most importantly for our analysis, the LLVM IR has explicit instructions marking the reading and writing a given variable. This implies that it is easy to distinguish getter functions (which only read the variable, hence are seldom focal methods under test) from mutator functions (which write to a given variable, hence are often good candidates for serving as a focal method under test). Secondly, LLVM IR disambiguates higher level programming constructs like function overloading. Indeed, many high level programming languages have the concept of function overloading, where the same function name can have different signatures for the same function name. It is not always apparent which overload of a given function will be used and it is therefore very hard to properly disambiguate what will happen. This in contrast to LLVM IR where every function name is unique, which is a clear benefit when establishing fine-grained test-to-method relationship.

LLVM obviously also comes with some disadvantages. First of all, the LLVM files can only be linked with other LLVM files, which on its own implies that we will need the source of every library used within the project. Secondly, LLVM IR suffers from a slight loss in context. Some high-level code constructs are not present in LLVM IR code as these are not required for the optimisation for which LLVM IR is intended. One example –importantly for our analysis– is the loss of public or private distinction.

4. Approach

Figure 1: Schematic Representation to Identify Focal Methods



The following section explains the approach used in the process of identifying the focal methods, as shown in Figure 1. The approach starts from an LLVM IR representation of the project, and results in a mapping of test methods to their corresponding focal methods under test.

4.1. Extracting Access Modifiers

As LLVM IR has no information about the access modifiers, we need to extract it beforehand. For this, we use the “clang-query” tool. For each file we store all its methods together with its

access modifier. We use this map to identify whether a method is `public` or `private` and how we need to act on it.

4.2. Identifying Tests

There are three major types of methods we consider for this analysis, test, assertion and source methods. A test method is distinguished from a source method by the naming convention used by the testing framework, which can simply be passed as an argument by the user. This is a necessary language dependent link as there is no fail proof way to distinguish test functions from tested functions.

Once the test methods are distinguished from all other methods, we enumerate all test methods. For each method we extract all relevant statements, in particular including all function invocations (where overloaded functions are disambiguated) and all instructions related to memory modifications.

4.3. Identify Assertions

To differentiate between assertions and source methods, we require a secondary input from the user, being the naming convention for assertions. These are normally implied by the testing framework used. This is usually a common prefix given to all assertion functions, eg. `assert` <assertion type> or use the class in which all assertions are defined, such as `AssertionResult` for the `GTest` framework.

4.4. Extract Variable Under Test

Before we can identify the focal methods for each test, we need to identify the variable under test (VUT). These are the values that are being verified in the assert statements, which we can extract these from the LLVM IR. Often these assert statements compare the VUT against a constant. Then it is easy to know which one of the two is our actual VUT. Other times, it is less obvious, as it evaluates against a variable from another class, In the latter case, we simply track both variables.

4.5. Identify Focal Methods

We identify the focal method by tracking the VUT throughout the invoked methods. During this process, any modification of the memory of the tracked variable will be considered a *mutator* function, hence a candidate focal method under test. It is now that the differentiation between public and private methods becomes important. We know that all candidate focal methods within the scope of the test suite are public. If one of these methods directly changes the VUT, then we label that method as a mutator. If the method does not directly change the VUT, but only invokes one or more public methods, then we don't descend deeper into the call tree according to the "no eager tests" assumption. If the method invokes one or more private methods, then we need to analyse these private method, as one of them can become the focal method, as in principle we cannot call private methods directly. We inspect all private methods

within private methods, but not public methods within private methods, as each public method should be tested by its own test.

For our analysis to work with libraries for which the source code is not given, we define a third class of invocations: *uncertain*. Method invocations labelled uncertain indicate that the definition of the method is not known due to absence of the actual implementation. Our approach resolves this by marking the function as being a potential focal method, but not the only focal method. The implication for a mutation analysis is that we will have slightly more tests linked to a mutant than when we would know the accessor modifier of the method.

4.6. Filter

Finally, an optional focal method conformance filter can be used. Without a means for filtering, functions defined within language specific libraries would be considered as being potential focal methods, which greatly reduces the effectiveness of the tool. For example, in C++, an assignments of string would be replaced by an assignment function defined within the standard C++ library. This function will never be the intended function under test, but considering a case where strings are compared in an assertion, the last function that will be used to assign said string to a variable will be this string assignment function defined within the std namespace. To prevent this function from being marked as the focal method, analysis of C++ code would benefit greatly from having the std namespace filtered from the set of focal methods. In our analysis, we would then still consider std library functions as functions leading to a mutation, but not as the focal methods itself.

5. Proof-of-Concept and Findings

We did a preliminary investigation with our proof-of-concept tool on an extended version¹ of the Stride project (version 1.18.06²). The details of which are in Table 1.

Table 1
Details Stride Project

Lines of code	3,776,986
Number of functions	54,811
Number of test functions	222

This project was chosen because it is written in a layered manner, where only the most outer layer is accessible. The classes located on the most outer layer are responsible for all classes directly below them, the classes below the outer classes will be responsible for the classes below those and so on. This structure is interesting, as it implies that test cases testing lower layers must traverse several other classes before arriving at the actual mutation which we want to test, making it an ideal candidate for our approach, as the method under test of those tests will be a private function.

¹<https://github.com/larsvanroy/stride>

²<https://github.com/broeckho/stride>

A manual inspection of the test code revealed that of the 222 tests, 77 serve as utility tests, 59 as i/o tests, 61 as population and generation tests and 25 as scenario tests. We list these in Table 2. Note that considering the size of the project, the test size may seem very limited, however, many of the functionalities part of the project are not testable on their own (as one would do with a unit test), due to the design choices made.

For now, we focused on the population and generation tests, as these adhere to the “no eager tests” assumption. More importantly, these 61 tests are tests in which the method under test is a private function in a lower layer. This selection allows us to more precisely determine the percentage of focal methods we can correctly identify in case a private method is the method under test. If we would not make this selection, the majority of the tests would be tests where the focal method is public. These tests are less interesting for our approach as we already demonstrated that the approach is capable of detecting the mutations in such instances.

Furthermore, a part of the utility tests were tests in which an inspector method was tested. Such tests directly test getters and might not even contain any mutations at all. Our approach cannot currently identify such tests correctly. We also disregarded scenario tests, as our approach is specifically intended to identify the method under test in unit tests, as scenario tests often test a list of methods, rather than one single method. We will have to investigate later whether our approach can be used for scenario tests.

Table 2

Test classification Stride project

Utility tests	77
Input/Output tests	59
Population and generation tests	61
Scenario tests	25

For 47 of our 61 tests, we identified the correct focal methods. Of the remaining 14 tests, 8 were misclassified because they corresponded with “no throw assertion tests”. Such tests only confirm that no exception is thrown when the object under test is manipulated. For the remaining 6, we identified the wrong focal method because the variable under test (VUT) where manipulated via pointers and this was not yet included in our LLVM IR Analysis.

5.1. Current Limitations

Library functions will be labeled as uncertain. This can negatively impact the speedup of a mutation testing analysis using focal methods.

Inspector test are currently not supported. We should be able to remedy this by taking into account that when no mutators are present, the last accessor method of the VUT, i.e. the inspector method should be considered as the focal method.

No throw assertion tests are currently not supported as they do not have actual VUTs. We should however be able to detect these scenarios. We will need to investigate how or which mutants we best link to them.

Pointer support in LLVM is something we have not yet implemented into our detection algorithm. Adding this should allow us to more accurately detect the focal methods.

5.2. Future Work

This proof-of-concept illustrates that it is feasible to implement the focal method under test heuristic on top of the LLVM compiler infrastructure. However, some extensions are needed to improve the accuracy of the tool. Most importantly, we need to expand the analysis to deal with assertions and pointers. Next, we need to automatically assess whether a given test case indeed satisfies the “no eager test” assumption. If we see that the assumption does not hold for a given test case, we should just revert to full scope mutation analysis. Last but not least, we need to test the tool on a larger scope of projects to see whether we can achieve similar speed-ups as what we estimated on java projects.

Acknowledgments

This work is supported by (a) Research Foundation - Flanders (project number 1SA1521N); (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

- [1] S. Ng, T. Murnane, K. Reed, D. Grant, T. Chen, A preliminary survey on software testing practices in australia, in: *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, IEEE Press, Piscataway, NJ, USA, 2004, pp. 116–125.
- [2] V. Garousi, J. Zhi, A survey of software testing practices in canada, *Journal of Systems and Software* 86 (2013) 1354–1376.
- [3] G. J. Myers, *The art of software testing*, 1979.
- [4] B. Adams, S. McIntosh, Modern release engineering in a nutshell—why researchers should care, in: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, IEEE Press, Piscataway, NJ, USA, 2016, pp. 78–90.
- [5] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, *IEEE transactions on software engineering* 37 (2011) 649–678.
- [6] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, M. Harman, Mutation testing advances: An analysis and survey, *Advances in Computers In Press* (2019) —. doi:10.1016/bs.adcom.2018.03.015.
- [7] T. T. Chekam, M. Papadakis, Y. L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017, pp. 597–608.
- [8] L. Zhang, D. Marinov, S. Khurshid, Faster mutation testing inspired by test prioritization and reduction, in: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, 2013, pp. 235–245.

- [9] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. Le Traon, J.-Y. Marion, Sound and quasi-complete detection of infeasible test requirements, in: *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, IEEE, 2015, pp. 1–10.
- [10] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, *Information and Software Technology* 54 (2012) 915–932.
- [11] D. Holling, S. Banescu, M. Probst, A. Petrovska, A. Pretschner, Nequivack: Assessing mutation score confidence, in: *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*, IEEE, 2016, pp. 152–161.
- [12] M. Ghafari, C. Ghezzi, K. Rubinov, Automatically identifying focal methods under test in unit test cases, in: *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, IEEE, 2015, pp. 61–70.
- [13] S. Vercammen, M. Ghafari, S. Demeyer, M. Borg, Goal-oriented mutation testing with focal methods, in: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 23–30.
- [14] M. Ghafari, K. Rubinov, K. Pourhashem, M. Mehdi, Mining unit test cases to synthesize API usage examples, *Journal of Software: Evolution and Process* 29 (2017).
- [15] A. Van Deursen, L. Moonen, A. van den Bergh, G. Kok, Refactoring test code, in: *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.
- [16] C. A. Lattner, LLVM: An infrastructure for multi-stage optimization, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2002.