Analysis of Focal Methods using Intermediary LLVM

Lars Van Roy
dept. of Mathematics and Computer Science
University of Antwerp
lars.vanroy@student.uantwerpen.be

January 6, 2021

Abstract

Iterative software development depends on continuous testing, as to provide rapid feedback about code issues throughout the development process. Test-to-code traceability links allow for better understanding between test and code artefacts. These links allow for developers to keep test code synchronised with tested code, as well as facilitating processes such as test-driven development. This paper will provide an overview of a means to automatically identify the focal method in unit test cases via analysis of intermediary LLVM. This is done in an attempt to provide a language independent way of this analysis. Validation of this approach on the Stride project shows a correct identification of focal methods in 83% of the cases.

1 Introduction

Agile development processes require unit tests to be written alongside source code, so that these can continuously be analysed, as is relied upon by many agile practices [6]. To fully realise the benefits of these tests, a good understanding of the link between the tests and the methods they intend to test is vital. These links, called test-to-code traceability links, allow us to keep test code up to date with regards to potential modifications to the source code itself, as well as allow for easier program comprehension.

One of the major benefits of test-to-code traceability links is the option to use these in both directions, which allows for developers to easily understand why certain tests fail. For processes such as test-driven development, it also facilitates an easy overview of the remaining non functional functionalities, by tracing the failing tests to their corresponding methods. They allow developers to start from a certain function, and check whether or not a certain aspect of a method is tested, as well as simply verifying whether or not a method is tested at all.

To create test-to-code traceability links, we will use the concept of focal methods. A focal method is the method that modifies the state of an object in a way that is then verified at the end of the test. This is different to general test coverage, as we now only consider the main focus of a test, as being a method under test, rather than considering every single function used within a test scope.

It has been proven that using these traceability links can also enhance other processes that would normally not use test-to-code traceability links, such as mutation testing [13]. In mutation testing we aim to quantify the fault-detection capability of a test suite by intentionally introducing faults, also called mutants, into the code. We then execute the tests as to verify whether or not the test suite fails. These links can allow for a specific approach where all tests related to a specific mutated method are executed, rather than an entire test suite for each potential mutation.

There has been an effort to do this in a manual way, either via developers manually maintaining traceability links, or via means of naming conventions. In most cases, these have been at the class level where the number of links was more maintain-

able and limited, however, this is not a desired approach, as the overhead of this process towards developers would be too large.

The need for a new methodology is due to a lack of language independent approaches. Language independent approaches would be valuable as this would allow us to automatically analyse projects, independently of the used programming language or programming conventions. It would be applicable to a majority of cases, rather than an approach for specific cases, which might be slightly more efficient, but will also have a significantly smaller use case. It can also serve as a great motivation to further build upon this, as the increase in use case, would also increase the value of further extensions.

2 Related work

As to prevent manual verification there has been research on automated approaches for test-to-code traceability links, as is documented by Prizi et al. [10] as a general overview. Most of this research has been focussed on linking test methods to the corresponding test classes [3, 2, 4, 8, 11, 12] but not as much work has been done on the method level [1, 5, 7, 14]. Furthermore, all research shown within this overview failed to give a methodology, capable of applying the method in a language independent manner. The approach given within this paper, will be based upon the findings described in the language dependent analysis by M. Ghafari et al. [5] but will do so in a language independent way.

3 LLVM IR

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 optimized using an aggressive multi-stage optimization provided within LLVM to then be converted to machine-dependent assembly language code [9].

What we are interested in for this research, is 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. It's main aspect that distinguishes it from assembly languages, is that it makes the assumption that there are infinite temporary registers.

The reason LLVM IR is used by several compilers, is because the capabilities it offers with regards to optimization. LLVM IR is mainly developed for use with clang, but has been introduced in several other compilers as a consequence of its effectiveness.

This intermediary language is used for this analysis, as this is a language that is capable of representing all major programming languages. It will therefore make the analysis fully language independent, with the only requirement of there existing a compiler that compiles the original language to LLVM IR. For the majority of the commonly used programming languages this will be the case.

It furthermore has the major advantage of being very limited in the instructions it contains. This means that there is an explosion in lines, but it also implies that analysis can be very simple, as compared to high level programming languages. The statements used within LLVM IR code are clearly defined and there is little to no need for disambiguation between statements, which is a major factor of analysis via forms of abstract syntax trees.

It does have the disadvantage that the LLVM optimization step occurs before the linking phase. This means that every file will need to be compiled on its own, and linked manually afterwards. 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. If not, the analysis will still work, but it's conclusion will be less valuable as there will be data missing.

4 Identifying focal methods

The following section explains the methodology used in the process of identifying the focal methods. The approach will start from an LLVM IR representation of the project, and will result in a mapping of test methods to their corresponding focal methods.

4.1 Motivation

Basing the approach on conventions such as naming conventions will not always give desired results. While it might be true that many projects do use a naming convention, in most software development companies these are often not the same nor are they always free to choose by the developers. Many other tools such as test frameworks already imply naming conventions on their own, which makes it nearly impossible to define an approach that is usable in all cases. If no naming conventions are implied by other tools, and we use the general approach of the function name with test appended or prepended, there is still the issue of tests and functions not being a one to one mapping. It is possible that a function needs to be tested in multiple scenarios and it is therefore impossible, but also impractical, to just name the tests according to the function under test. One should try to name the function according to the methods under test as well as the scenario that is being evaluated.

Other approaches exist which specifically analyse the test code itself. There is the approach of directly considering the last call inside a test body to be the function under test, but this is ineffective in languages such as c++ and java. In these languages it is often the case that the method under test is a method modifying a private variable which will then need to be accessed via an inspector method to be compared against the oracle. In this case it is not the last call that is the method under test, but rather we are looking for the last function that mutated the object or variable under test.

In our approach, we do not assume a naming convention to be followed by the developer, nor do we assume that the focal method must have a certain place within the function body. We will consider the variables used within the assertions, trace their evolution throughout the test method, and find the method that modified the value of the test variable last.

4.2 LLVM IR analysis

There are three major types of methods we will consider for this analysis, being test, assertion and source methods. A test methods will be 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. By allowing this to be an argument from the user we remain the language independent aspect of our analysis.

With the test methods distinguished from all other methods, we must now analyse those test methods. For each method we'll extract all relevant statements, in particular including all invocations and all instructions related to memory modification. For each of the used methods, we will differentiate between assertions and source methods. To do this in a correct way, we will require a secondary input from the user. We can now analyse the source functions found within the test code, followed by the source functions found within the previous source functions.

To limit the execution time of the analysis, a tertiary depth parameter can be requested from the user, implying the maximal depth for the function evaluation. Throughout the remainder of this paper, a maximal depth is assumed to be given.

Example 1 represent a very simple environment where there is a profile class with corresponding methods, and a single test testing the *setFirstName* method. For the test to work, an instance of the *Profile* class is needed, as is instantiated on line 16

in the test body. This line is then mutated on line 17 to be retrieved again on line 18. The resulting value will then be compared on line 19.

Our approach will start by distinguishing the test method defined on line 15 from the other methods (note that there is no notion of classes within LLVM, this means that class methods are indistinguishable from other methods). Within this test method's body it will evaluate the invoked methods on line 16, 17 and 18. It will not consider the assertion function on line 19, as it is able to distinguish assertions from other methods.

4.3 Focal methods

Unit test cases often consist of a series of method invocations, ultimately leading to a variable being compared against an oracle. In most cases, most of the used methods are ancillary, and are purely used to get the environment prepared for the actual method under test. Once the method under test has been applied on the environment, there might still be a need for additional method invocations, to get the desired information out of this environment. It is therefore important to differentiate between functions in these three different phases of a test method.

In our approach, we will not differentiate between methods used within the initial setup, and the actual method under test, but we will try to differentiate between the ancillary methods used within the comparison phase, and the methods used to modify the environment. A method that is used to inspect the state of the environment, is called an *inspector*. These methods will be considered to trace the evolution of our test variables, but these will never be considered as being the focal methods of a test suite. A function that does mutate the environment is called a *mutator*, and these will be considered as potential focal methods.

Example 1: examplary test environment

```
1  class Profile
2    firstName
3    lastName
4
5    method setFirstName(profile, name)
6      store name in profile.firstName
7    endmethod
8
9    method getFirstName(profile)
10     var ← load firstName from profile
11     return var
12   endmethod
13 endclass
14
15 method testSetFirstName()
16   p ← Profile()
17   p.setFirstName("Ani")
18   name ← p.getFirstName()
19   assertEq(name, "Ani")
20 endmethod
```

Function that might mutate a variable, but are impossible to know for sure are marked as being *uncertain*. A need for uncertain can occur when part of a project is missing due to absence of source code for a used library. In this case, functions that use undefined functions will be marked as being potential mutators, as there is no way to tell whether or not the passed variables actually got mutated within the called functions. A second case in which uncertainty might occur, is whenever the max depth of our analysis is too low. In this case it might be that we encounter a function that was not analysed within the first analysis phase. In this case we will mark the function as uncertain, until an actual mutation is found.

In terms of the analysis, uncertain functions will be considered as being a potential focal method, but our analysis will not stop when one is found. It will simply carry on, until the end of the function body is reached, or until an actual mutation occurs.

Finally, an optional focal method conformance

filter can be used, which would be a filter to which focal methods must not conform. This is introduced as the language independence has significant effects on the evaluation of focal methods. 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.

When going back to the environment represented within Example 1, we can see that we have one assertion, being the invocation on line 19 within the *testSetFirstName* method. This method has two variables being used, being the constant "Ani", which will not lead to any mutations at all, and the name variable. Tracing this name variable upwards, will lead us to the *getFirstName* method, but analysis of the resulting code, will not show any statements modify memory and it will therefore be flagged as being an inspector. We will also detect that the profile variable was used, which will become our new tracked variable. this very same variable will be used on line 17, and analysis of this function body will show a mutation, being the store on line 6. This means that this function will in fact be flagged as being a mutator, which will end the evaluation for function *testSetFirstName* with the resulting mutator being *setFirstName*.

5 Case study

In this section, we will give an example case in which the tool was applied as well as the results obtained from it. Considering the obtained results, we will give an overview in what way these results satisfy our original goal for this research.

5.1 Stride

To evaluate the tool, it was applied on a forked and extended version of the original Stride project (version 1.18.06[1]), which is a project written in C++. This extended version [2] includes several additional classes, along with corresponding tests. The tool was applied to the entirety of the test suite, and the used LLVM IR contained compiled versions of all used libraries by the project. This resulted in an LLVM file of which statistics are listed within Table 1. The LLVM IR was generated using the Clang compiler[3](version 10.0.0).

lines of code	3,776,986
number of functions	54,811
number of test functions	222

Table 1: Statistics for the generated LLVM IR code.

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 will be 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.

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, due to the design choices made, and are tested in a more grouped way.

5.2 Research questions

A few questions need to be answered, to quantify the usefulness of the approach for the given test case. Below is an overview of the questions we attempted

[1] https://github.com/broeckho/stride
[2] https://github.com/larsvanroy/stride
[3] https://clang.llvm.org/

to answer using the results of our simulation, alongside on overview of why this question needs to be asked, and how we intend to find a solution.

Question 1: *To what extent are can we correctly identify the focal methods?*
Motivation. Given that the tool main goal is to correctly identify the focal methods for a given project, we want to know the measure in which it is capable of doing so. It is unlikely that every single focal method can correctly be identified due to several reasons. These reasons could be a lack of a focal method under test, lack of compared variables (e.g. assert that no throw occurred) or the last mutation call was not the focal method of the test method.
Approach. We consider the precision and recall of several simulations using different parameters. The precision will be used to determine the percentage of selected focal methods that are actual focal methods. The recall will be used to determine how many of the focal methods we actually find. To allow proper comparisons between approaches, the harmonic mean from both the precision and recall will be considered, computed using the formula below.

$$h - mean = \frac{2 \times precision \times recall}{precision + recall}$$

Question 2: *In what way is the depth parameter related to the gain of the tool?*
Motivation. The depth parameter is intended to increase the precision of the result, but might also have a negative effect on the recall. It will furthermore have a direct effect on the evaluation time of the analysis. It is important to evaluate the negative effects on the resulting recall to estimate the effect of undefined functions. The execution time is also relevant, as a percentage wise increase in evaluation time will be significant in larger projects.
Approach. To test this we will consider the accuracy, recall and evaluation time for all depth values ranging from 0 to 4. By considering the differences between these different simulations we can get an idea of the effect of the depth parameter and from those we can then derive the actual gain of increasing/decreasing the depth.

Question 3: *In what way can we mitigate the result of language independence?*
Motvation. The analysis is language independent, and this obviously has an effect on the result. No knowledge of language specific functionalities often results in incorrect determination of focal methods, as these functions will never be the function under test, but might be marked as being so. The effect is important to consider in comparison with other tools.
Approach. To evaluate this, we will consider two different simulation runs, one without a filter and one with a filter, filtering out all functions part of the std namespace. Comparison of the obtained harmonic mean between the two simulations will give us an indication in the gain from removing language specific functionalities from the potential focal methods.

5.3 Results

Table 2 visualises the results obtained from the simulation. For each of the considered depths it gives the precision, recall, mean and time it took to run the simulation. For each depth we ran two different simulations, one without a filter and one with a filter, filtering out just the methods part of the std namespace. Note that we only gave simulations up to depth 4, as this was the maximal depth to which the considered tests reached. Depths beyond 4 would have an increase in simulation time, but the precision, recall and mean would not change.

Question 1: *To what extent are can we correctly identify the focal methods?*
Over all 10 simulations, the lowest encountered recall is 76%, which indicates that the approach is capable of identifying focal methods. The precision however is not good for the lower depth simulations. This is expected. As low depth means that many functions

filter applied	parameter	depth				
		0	1	2	3	4
yes	precision	0.25	0.32	0.49	0.51	0.72
	recall	0.79	0.86	0.83	0.83	0.83
	mean	0.38	0.47	0.62	0.63	0.77
	time	20.96 s	29.23 s	31.69 s	37.36 s	41.83 s
no	precision	0.25	0.29	0.48	0.49	0.65
	recall	0.79	0.76	0.76	0.76	0.76
	mean	0.38	0.42	0.59	0.60	0.70
	time	20.09 s	28.83 s	31.76 s	37.18 s	42.71 s

Table 2: Statistics for the generated LLVM IR code.

will not be known in the focal method phase of the approach. It will mark most of the used functions as being uncertain mutators, which causes the recall to be high, but the precision to be low. The precision increases as the depth goes up and at depth 4, for the simulation that applied the filter, we can see a precision of 72% with a harmonic mean of 77%. This clearly indicates that we are in fact capable of doing meaningful analysis given a test suite and a project.

The reason the precision is not higher than it is, is because our language independent approach does not allow us to distinguish between expected value and value under test. this means that we often get two focal methods for a single assertion, where most cases will only have one variable that was actually under test. An additional parameter indicating the parameter location within an assertion (e.g. the asserted variable is the second variable in an assertion) might resolve this for projects in which a standard is used for this, but this once again requires developers to uphold a convention, which is one of the things we intended to avoid using this approach.

The cases in which the focal method was not found at all, were mostly cases where a no throw assertion was used. In these cases the tool has no capability of detecting what the function under test was, as there is no variable to track through the code. Other cases where the focal method was not found, were cases where an object was copied into a shared pointer, this operation is assumed to be a mutation by the tool, and it blocked the path to the actual mutation, which occurred just before it. Finally, cases where inspector methods are tested, will never return the correct method and were therefore also incorrectly classified.

When specifically considering the recall for the simulations with no applied filter, we can see that it does in fact drop when we increase the depth. This means that there is an std function (which can be assumed, as this decline in recall did not occur when the filter was applied) that is considered by the approach as being a mutator, that occurred before the actual focal method. This means that while it did return the correct focal method for depth 0, from depth 1 and onwards, it will stop at said std function and it will never reach the actual focal method.

Question 2: *In what way is the depth parameter related to the gain of the tool?*
When comparing the simulation for depth 0 compared to depth 4, we can see a definite increase in the mean parameter, mostly resulting from an increase in precision. This is expected to happen, as a higher depth, means less uncertain methods. An method that no is no longer uncertain must either become an inspector or a mutator. In case it becomes a mutator, it means that it will become the focal method for the considered variable of the assertion, and no further functions will be considered.

In case it becomes an inspector, we will simply carry on to the next relevant line of code, and the function will not be considered as being a focal method, where it did while it was a mutator.

While this increase in mean is exactly what we wanted, we must consider the increase in simulation time. Up to a depth of 4, the simulation time already doubled. This is a clear indication that the depth should be limited based upon weighted considerations. In cases where accuracy is critical, a higher depth is definitely in order, but where a slightly lower accuracy in return for speed is desired, a lower depth might be considered.

Question 3: *In what way can we mitigate the result of language independence?*
When comparing the results from the simulations where the filter was applied, to the results where the filter was not applied, we can definitely see an improvement in both precision and recall. Note that the mean is higher or equal for all simulations of equal depth, in favour of the simulation using a filter. This indicates that the use of a filter definitely has a positive effect on the quality of the results.

6 Threats to validity

We note a a number of threats to the validity of the found results.

A primary remark should be that we did not consider multiple languages, even though this is a language independent tool. This remark is correct, in the sense that we should have performed analysis on bigger and more diverse projects, although that the analysis is fully language independent. We analysed a semi large LLVM file which contained the majority of the possible LLVM statements. The approach managed to analyse this correctly, and gave results based upon this. In this approach, no language specific components should have been present, other than the input from the user itself, and this input is possible for any chosen programming language.

Secondary, we need to consider that we did not write all of these tests and there is a human error present in the sense that we manually determined the correct result of the tool. However, considering that some of the tests were written ourselves, we believe that the we correctly identified the actual goals of the tests, and thereby correctly assumed the intended focal method.

Other than the previous two remarks, this analysis is not dependent on any conventions nor did we consider the help of any outside tools. We should therefore not undergo any accumulative error from such sources.

7 Conclusion

In this paper, we presented a new automated method to determine test-to-code traceability links using focal methods.The focal methods allowed us to determine a link from test code to their tested methods.

We were able to show that this approach is indeed capable of detecting focal methods, by applying the approach to Stride. Considering this example, we saw that 83% of the focal methods was indeed correctly determined, with a precision of 72%.

While the results of the experiment were promising, there is still need for further improvements. For this paper, we only considered one project, which only covered one language. This being a language independent approach, further evaluation of different languages should be performed, before fully concluding that this approach is capable of analysing all major programming languages in a language independent way.

References

[1] Philipp Bouillon et al. “EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors”. In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Giulio Concas et al. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2007, pp. 101–104. ISBN: 978-3-540-73101-6.

[2] V. Csuvik, A. Kicsi, and L. Vidács. "Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability". In: *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*. 2019, pp. 29–36. DOI: 10.1109/SST.2019.00016.

[3] Viktor Csuvik, András Kicsi, and László Vidács. "Evaluation of Textual Similarity Techniques in Code Level Traceability". In: *Computational Science and Its Applications – ICCSA 2019*. Ed. by Sanjay Misra et al. Cham: Springer International Publishing, 2019, pp. 529–543. ISBN: 978-3-030-24305-0.

[4] M. Gethers et al. "On integrating orthogonal information retrieval methods to improve traceability recovery". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 133–142. DOI: 10.1109/ICSM.2011.6080780.

[5] M. Ghafari, C. Ghezzi, and K. Rubinov. "Automatically identifying focal methods under test in unit test cases". In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2015, pp. 61–70. DOI: 10.1109/SCAM.2015.7335402.

[6] T. D. Hellmann et al. "Agile Testing: Past, Present, and Future – Charting a Systematic Map of Testing in Agile Software Development". In: *2012 Agile Conference*. 2012, pp. 55–63. DOI: 10.1109/Agile.2012.8.

[7] Victor Hurdugaci and Andy Zaidman. "Aiding software developers to maintain developer tests". In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE. 2012, pp. 11–20.

[8] A. Kicsi, L. Tóth, and L. Vidács. "Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability". In: *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. 2018, pp. 8–14.

[9] Chris Arthur Lattner. "LLVM: An infrastructure for multi-stage optimization". PhD thesis. University of Illinois at Urbana-Champaign, 2002.

[10] R. M. Parizi, S. P. Lee, and M. Dabbagh. "Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts". In: *IEEE Transactions on Reliability* 63.4 (2014), pp. 913–926. DOI: 10.1109/TR.2014.2338254.

[11] Abdallah Qusef et al. "Recovering test-to-code traceability using slicing and textual analysis". In: *Journal of Systems and Software* 88 (2014), pp. 147–168.

[12] Bart Van Rompaey and Serge Demeyer. "Establishing traceability links between unit test cases and units under test". In: *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE. 2009, pp. 209–218.

[13] Sten Vercammen et al. "Goal-Oriented Mutation Testing with Focal Methods". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 23–30. ISBN: 9781450360531. DOI: 10.1145/3278186.3278190. URL: https://doi.org/10.1145/3278186.3278190.

[14] Robert White, Jens Krinke, and Raymond Tan. "Establishing Multilevel Test-to-Code Traceability Links". In: *42nd International Conference on Software Engineering (ICSE'20)*. ACM. 2020.