UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INFORMATICS INSTITUTE
BACHELOR OF COMPUTER SCIENCE

KARINA GABIN MINUZZO

Prototype for Analysis of Different Coverage Criteria of Object Oriented Code

Bachelor Thesis

Prof. Dr. Érika Fernandes Cota
Advisor

Porto Alegre
2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Opermann
Pró-Reitora : Valquíria Linck Bassani
Diretor do Instituto deInformática: Prof. Luís da Cunha Lamb
Coordenador do curso de Ciência da Computação: Carlos Arthur Lang Lisboa
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGMENTS

I would like to thank, in first place, to my parents who always supported me and inspired me to search for knowledge and be dedicated to continuous learning. I also would like to thank them for all the good moments we spent together and all dedication they had with me.

I would like to thank to the friends and family that make life easier and softer enjoying together and celebrating, not forgetting the difficulties when I could count on them to be on my side. A special thanks for the friends I met during the course and who I believe will be always important on my life.

Also, I want to thank all the professors I had at UFRGS that were always dedicated and concerned with maintaining excellence. And last I would like to thank to all that made my internship to Germany possible and pleasant.

CONTENTS

CONTENTS..........3
RESUMO..........5
ABSTRACT..........6
LIST OF FIGURES..........7
LIST OF TABLES..........9
LIST OF ACRONYMNS AND ABBREVIATIONS..........10
1 INTRODUCTION..........11
1.1 Motivation..........11
1.2 Structure..........12
2 BACKGROUND..........13
2.2 Graph Coverage for Source Code..........14
2.2.1 Structural Coverage Criteria..........16
2.2.2 Data Flow Coverage Criteria..........17
2.3 LLVM..........20
2.3.1 LLVM Intermediate Representation..........21
3 IMPLEMENTATION..........23
3.1 The Prototype Tool..........23
3.1.1 Input Parsing..........23
3.2 Structure of the Proposed Approach..........28
3.3 Implementation of the Coverage Criteria..........30
3.3.1 Find Simple Path..........30
3.3.2 Prime Path..........31
3.3.3 Is Valid Path..........32
3.3.4 Complete Round Trip..........33
3.3.5 All DuPair..........33
3.3.6 DuPath..........34
3.3.7 All DefPath..........35
3.3.8 All UsePath..........36
3.3.9 DefClear..........38
3.4 Algorithms Analysis..........38
4 EXPERIAMENTAL RESULTS..........39
4.1 Running Example..........39
4.1.1 Input Program..........40
4.2 File Processing..........41
4.3 Test Execution..........42
4.3.1 Prime Path..........42
4.3.1.1 Prime Path Criteria Test Run..........43
4.3.2 Du-Path..........44
4.3.2.1 All Def-Path Test Run..........45
4.3.2.2 All Use-Path Test Run..........46
4.4 Validation..........47
4.4.1 Program Execution..........47
4.4.1.1 First Example..........47

4.4.1.2 Second Example…………………………………………………………………..
4.4.1.3 Third Example……………………………………………………………….........61
4.4.1.4 Fourth Example……………………………………………………………..........69
5 FINAL REMARKS……………………..……………………………………….........77
REFERENCES…………………………………………………………..…….…..........78

RESUMO

Softwares desempenham um papel importante facilitando, automatizando e controlando atividades essenciais ou interessantes aos usuários. Alguns deles muitas vezes são sistemas críticos, envolvendo riscos ao apresentar uma falha. A área de teste de software se apresenta como uma solução para a redução dos riscos de um sistema se comportar diferentemente do esperado.

Diante da crescente aplicação de métodos de teste em sistemas computacionais e visto sua essencialidade para a geração de softwares de qualidade, existe a necessidade de ferramentas capazes de auxiliar e automatizar este processo. Este trabalho consiste no desenvolvimento do cerne de uma ferramenta capaz de analisar diferentes critérios de cobertura de código orientado a objeto.O protótipo apresentado contém um conjunto de funções básicas e essenciais para a aplicação automática dos critérios de cobertura basedos em grafos a partir de uma linguagem intermediária que pode ser gerada a partir de diferentes linguagens de programação.

Palavras-chave: teste de software, automação de teste, cobertura de código

Application for Analysis of Different Coverage Criteria of Object Oriented Code

ABSTRACT

Software performs an important role facilitating, automating and controlling essential or just interesting activities to their users. Some of them are critical systems, involving risks in case of failure. The testing field works as a solution to reduce risks of software non-expected behaviors.

Given the increasing use of test methods in computer systems and considering its essentiality for the generation of high quality software, there is a need for tools that can assist and automate this process. This work presents the prototype of a tool that helps the developer and the tester to analyze different graph-based coverage criteria that cannot be easily found in available tools. Such a prototype is the seed for the implementation of a more complete tool targeting the test of object-oriented code.

Keywords: software testing, testing automation, code coverage

FIGURES LIST

Figure 2.1 - Path Examples..14
Figure 2.2 - CFG fragment for the “if” structure without an else…………………………15
Figure 2.3 - CFG fragment for the “while” loop structure..15
Figure 2.4 - CFG fragment for the “for” loop structure...16
Figure 2.5 - fnc_move_data32 for data flow example……………………………………18
Figure 2.6 - fnc_move_data32 CFG…..19
Figure 2.7 - Intermediate code of fnc_move_data32..22
Figure 3.1 - File Lifecycle Flowchart...24
Figure 3.2 - Parsing Intermediate File Flowchart..24
Figure 3.3 - Graph Creation Flowchart...………….........25
Figure 3.4 - Set graph variables flowchart..26
Figure 3.5 - Set variables definitions flowchart...27
Figure 3.6 - Set variables uses flowchart..27
Figure 3.7 - Application Class Diagram...29
Figure 3.8 - Find Simple Path method...31
Figure 3.9 - Prime Path method…………………………………………………………...32
Figure 3.10 - Is Valid Path method...33
Figure 3.11 - Complete Round Trip method...33
Figure 3.12 - All Du-Pair method..34
Figure 3.13 - All Du-Path method..35
Figure 3.14 - Def-Path method..36
Figure 3.15 - Use-Path method...37
Figure 3.16 - Is Def-Clear method...38
Figure 3.17 - “findSimplePath” average execution time..39
Figure 4.1 - CFG of input program with defs and uses...41
Figure 4.2 - Program execution for Prime Path criteria. 43
Figure 4.3 - Prime path TR for “entry” to “for.end”……………………………………......44
Figure 4.4 - Program call for def-path method...45
Figure 4.5 - Program call for use-path method...46
Figure 4.6 - “isNewMonth” Source Code…………………………………………………48
Figure 4.7 - “isNewMonth” Intermediate Code...49
Figure 4.8 - “isNewMonth” correspondent CFG..50
Figure 4.9 - Execution of Simple Path method for “isNewMonth“ graph and considering the nodes “entry” and “if.end” 50
Figure 4.10 - Execution of Prime Path method for “isNewMonth“ graph, considering the nodes “entry” and “if.end” 51
Figure 4.11 - Execution of Complete Round Trip method for “isNewMonth“ graph, considering the node “if.end” 51
Figure 4.12 - Execution of Du-Pair method for “isNewMonth“ graph, considering the variable “daysOfTheMonth” 52
Figure 4.13 - Execution of Du-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth” ………………………………………………………...........52

Figure 4.14 - Execution of All Du-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth” 53
Figure 4.15 - Execution of All Def-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”..53
Figure 4.16 - Execution of All Use-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”..54
Figure 4.17 - “isNewMonth” Source Code...55
Figure 4.18 - “update” Intermediate Code……………………………………….........56
Figure 4.19 - “update” correspondent CFG……………………………………….........56
Figure 4.20 - Execution of Simple Path method for ”update” graph considering nodes “entry” and “if.end” 57
Figure 4.21 - Execution of Prime Path method for “update“ graph, considering the nodes “entry” and “if.end”……………………………………………………………………………58
Figure 4.22 - Execution of Complete Round Trip method for “update“ graph, considering the node “if.else”…………………………………………………………………………58
Figure 4.23 - Execution of Du-Pair method for “update“ graph, considering the variable “subject”…………………………………………………………………………………..59
Figure 4.24 - Execution of Du-Path method for “update“ graph, considering the variable “subject”…………………………………………………………………………………..59
Figure 4.25 - Execution of All Du-Path method for “update“ graph, considering the variable “subject”…………………………………………………………………………60
Figure 4.26 - Execution of All Def-Path method for “update“ graph, considering the variable “subject”…………………………………………………………………………60
Figure 4.27 - Execution of All Use-Path method for “update“ graph, considering the variable “subject”…………………………………………………………………………61
Figure 4.28 - “eraseMemory” Source Code……………………………………………...62
Figure 4.29 - “eraseMemory” Intermediate Code………………………………………….63
Figure 4.30 - “_eraseMemory” correspondent CFG………………………………………64
Figure 4.31 - Execution of Simple Path method for “_eraseMemory“ graph, considering the nodes “entry” and “for.end”………………………………………………………………64
Figure 4.32 - Execution of Prime Path method for “_eraseMemory“ graph, considering the nodes “for.body” and “for.end"………………………………………………………...65
Figure 4.33 - Execution of Complete Round Trip method for “_eraseMemory“ graph, considering the node “for.cond”…………………………………………………………65
Figure 4.34 - Execution of Du-Pair method for “_eraseMemory“ graph, considering the variable “i”…………………………………………………………………………………66
Figure 4.35 - Execution of Du-Pair method for “update“ graph, considering the variable “memorySize”……………………………………………………………………………66
Figure 4.36 - Execution of Du-Path method for “_eraseMemory“ graph, considering the variable “i”…………………………………………………………………………………67
Figure 4.37 - Execution of All Du-Path method for “_eraseMemory“ graph, considering the variable “i”……………………………………………………………………………67
Figure 4.38 - Execution of All Def-Path method for “_eraseMemory“ graph, considering the variable “i”……………………………………………………………………………68
Figure 4.39 - Execution of All Use-Path method for “_eraseMemory“ graph, considering the variable “memorySize”…………………………………………………………………68

Figure 4.40 - “getByte” Source Code……………………………………………………….69
Figure 4.41 - “getByte” Intermediate Code…………………………………………………70
Figure 4.42 - “getByte” correspondent CFG………………………………………………..71
Figure 4.43 - Execution of Simple Path method for “getByte “ graph, considering the nodes “entry” and “for.inc”……………………………………………………………………72
Figure 4.44 - Execution of Prime Path method for “getByte “ graph, considering the nodes “entry” and “for.inc”……………………………………………………………………72
Figure 4.45 - Execution of Complete Round Trip method for “getByte “ graph, considering the node “for.cond”…………………………………………………………..73
Figure 4.46 - Execution of Complete Round Trip method for “getByte “ graph, considering the node “for.body”…………………………………………………………..73
Figure 4.47 - Execution of Du-Pair method for “getByte “ graph, considering the variable “byte”………………………………………………………………………………………74
Figure 4.48 - Execution of Du-Pair method for “getByte “ graph, considering the variable “byte”………………………………………………………………………………………74
Figure 4.49 - Execution of Du-Pair method for “getByte “ graph, considering the variable “byte”………………………………………………………………………………………75
Figure 4.50 - Execution of All Du-Path method for “getByte “ graph, considering the variable “byte”………………………………………………………………………………75
Figure 4.51 - Execution of All Def-Path method for “getByte “ graph, considering the variable “byte”………………………………………………………………………………76
Figure 4.52 - Execution of All Def-Path method for “getByte “ graph, considering the variable “byte”………………………………………………………………………………76

TABLE LIST

Table 2.1 - Defs and uses at each node in the CFG for fnc_move_data32...................26
Table 2.2 - Coverage Criteria Example………………………………...........................27

LISTA DE ABREVIATURAS E SIGLAS

CFG	Control Flow Graph
TR	Test Requirement

INTRODUCTION

Software can be found everywhere in devices or systems that are part of society life. They define the behavior of network routers, smartphones, the Web and all infrastructure of modern life. Users do not want the software to present failures and quality engineering works in order to avoid such a situation.

Test methods in computational systems are essential for the generation of quality software, reducing the risk of failures and ensuring compliance and reliability of the developed system. To optimize this process, tools able to automatically assess the level of correctness of an application are used, what we call test automation [Offutt, 2008, p. 10]. This work consists in the development of a tool capable of automating the analysis of different graph-based coverage criteria thus helping the developer to ensure that the software works as expected relative to the requirements covered.

Test coverage criteria exists in order to optimize test methodology, they act focusing on the critical points of the program to be analyzed gaining time and resources, since exhaustive tests would cost much more to stress software and taking much longer.

1.1 Motivation

The goal of this study is to create a solution without the same limitations of the tools existent today. Current coverage tools are tied to parsers of specific programming languages such as Java or C++ or supported by specific Integrated Development Environments (IDEs) also tied to a single programming language. Thus, the dissemination of the testing coverage criteria depends on the re-implementation of the same algorithms for each programming language of interest in a given organization. Alternatively, a testing tool can receive a graph already extracted from the source code. In this case, the developer needs to either generate the graph by hand or to use different tools to achieve a single goal. Furthermore, coverage criteria currently

supported by available tools are usually the weakest ones, only executing simple methods of coverage, such as node coverage or edge coverage.

The proposed alternative is to implement the algorithms for graph-based coverage criteria using a common format that can be shared by many programming languages, that is, based on a structure that is not coupled to its source code. An existing environment that makes this approach possible is the LLVM libraries (The Clang Team, 200-?]), which provides a source and target-independent optimizer, along with code generation support for many popular CPUs. Although currently supporting C and C++ front-ends, the optimizer receives a generic and simpler structure that can be more easily implemented to other languages. Some initiatives in this direction already exist, such as the support to C# code (LLVM Project Blog, 2015).

Thus, the goal of the proposed tool is to be generic and able to apply more sophisticated methods of code coverage criteria directly to different programming languages.

1.2 Structure

This document is organized in order to first present a background and basic concepts of coverage criteria and the requirements needed to execute the application developed. Then the implementation will be covered, discussing each algorithm proposed for the criteria considered on this work. Afterwards will be presented the results obtained by the application itself being used over a real program being tested. Finally the final remarks and future work will be discussed in order to conclude this study.

2 BACKGROUND

In the testing role there are some terms for testing methods such as "exhaustive testing", "complete test" or "full coverage" that consider all the possibilities to be tested which would stress a software until its limit. The potential inputs for this kind of strategy is so large that tends to infinity.

Thinking about ways to improve testing within a limited schedule, we have the formal coverage criteria. Since generating this huge amount of data required by exhaustive testing would not be possible we need another way to guarantee the high quality and reliability of the software. The coverage criteria is supposed to satisfy this need, considering the test requirements, that are elements or software artifacts that a test case must satisfy or cover. Coverage criteria is defined by selecting the appropriated set of test cases following the rules necessary to meet the test requirements.

2.1 Graph Coverage Criteria

Oriented graphs are usually used as foundation for coverage criteria. They are obtained from given artifacts of the program which are under test. The control flow graph is an example that can be generated as an abstraction from source codes.

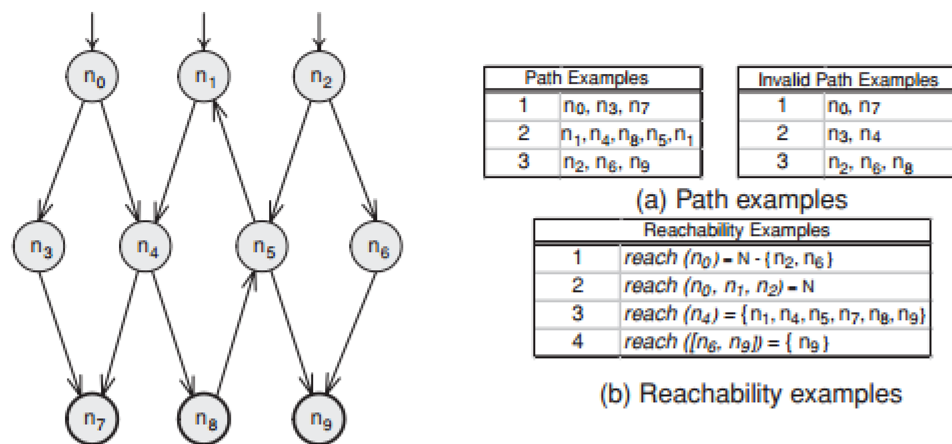
The graph coverage criteria analyzes a test set for an artifact based on how many paths correspondent to test cases can cover the graph.

A basic definition of graph is as the following [Offutt, 2008, p. 27]:

- set N of nodes
- set N_0 of initial nodes, where $N_0 \subseteq N$
- set N_f of final nodes where $N_f \subseteq N$
- set E of edges, where E is a subset of $N \times N$

Figure 2.1 presents an example of how control flow graphs are used in testing based on the paths reachability. The "Path Examples" listed are valid inputs because it is possible to follow the path from an initial node until a final node without interruption. "Invalid Paths Examples", on the other hand, are the ones that are not completely connected and have one or more nodes that cannot be accessed through the path.

Figure 2.1 - Path Examples



Path Examples	
1	n_0, n_3, n_7
2	n_1, n_4, n_8, n_5, n_1
3	n_2, n_6, n_9

Invalid Path Examples	
1	n_0, n_7
2	n_3, n_4
3	n_2, n_6, n_8

(a) Path examples

Reachability Examples	
1	$reach(n_0) = N - \{n_2, n_6\}$
2	$reach(n_0, n_1, n_2) = N$
3	$reach(n_4) = \{n_1, n_4, n_5, n_7, n_8, n_9\}$
4	$reach([n_6, n_9]) = \{n_9\}$

(b) Reachability examples

Source: [Offutt, 2008](p. 29)

A graph structure can be extracted from different software artifacts, such as class diagrams, use case diagrams or even a textual description. In all cases, once the graph is defined, the application of the criteria does not change. In the next sections the reachability coverage criteria will be discussed in the context of graphs extracted from source-code.

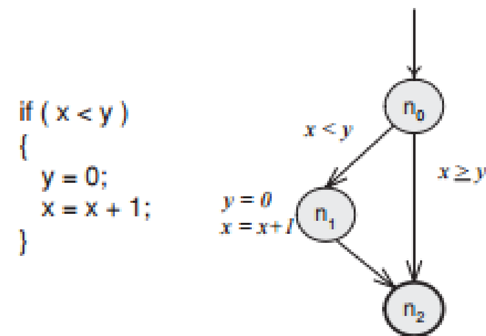
2.2 Graph Coverage for Source Code

When the focus of the test is the source code the most common approach used is based on the Control Flow Graph (CFG), according to Ammann and Offutt (2008). A

CFG consists of nodes and oriented edges, where the edges are associated with each possible branch in the program and the nodes represent sequence of statements, called basic blocks. A basic block is the biggest sequence executed from one point to another without interruption, that is, if the first line of a basic block is executed all the following lines of it will be called and executed too.

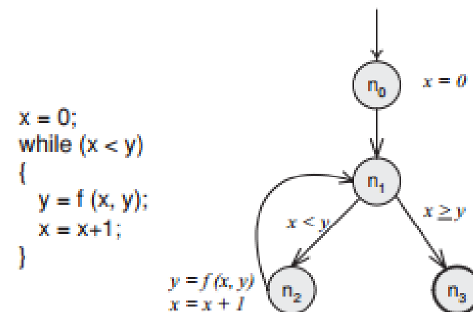
Figures 2.2 to 2.4 present CFG fragments that represent simple basic blocks of code.

Figure 2.2 - CFG fragment for the “if” structure without an else.



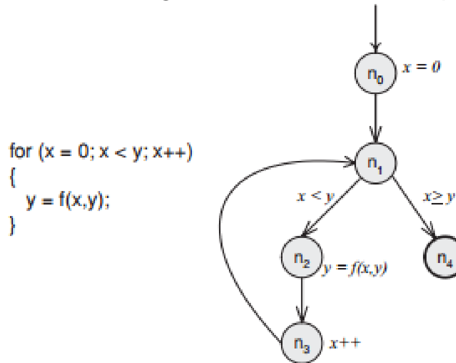
Source: [Offutt, 2008](p. 53)

Figure 2.3 - CFG fragment for the “while” loop structure.



Source: [Offutt, 2008] (p. 53)

Figure 2.4 - CFG fragment for the “for” loop structure.



Source: [Offutt, 2008] (p. 54)

2.2.1 Structural Coverage Criteria

The structural coverage criteria are defined on a graph just in terms of nodes and edges. We define a coverage criteria by specifying the set of Testing Requirements (TR) which are, for this kind of coverage, to visit every node and every edge in a graph. The first criterion, based on the nodes, has the concept we have to execute every statement in a program and is called node coverage. “Node Coverage (NC): TR contains each reachable node in G.” [Offutt, 2008, p. 33]. For example, for Figure 2.4, that is a CFG representation of a “for” structure, the TR for node coverage criterion would be {n_0, n_1, n_2, n_3, n_4} while a valid test path which covers the test requirements could be: [n_0, n_1, n_2, n_3, n_1, n_4].

The second criterion, based on the edges is usually implemented as branch coverage and is called edge coverage. “Edge Coverage (EC): TR contains each reachable path of length up to 1, inclusive, in G.” [Offutt, 2008, p. 34]. Using the same example of Figure 2.4, the TR for edge coverage criteria would be {(n_0, n_1), (n_1, n_2), (n_1,

n_4), (n_2, n_3), (n_3, n_1), (n_1, n_4)} while a valid test path which covers the requirements could be: [n_0, n_1, n_2, n_3, n_1, n_4].

Another coverage criterion based on touring the edges is the edge-pair coverage criterion, which requires that each path of length (up to) two be toured by some test path. This idea can be extended for any path length. This criterion can be defined as follows: "Edge-Pair Coverage (EPC): TR contains each reachable path of length up to 2, inclusive, in G." [Offutt, 2008, p. 35]. For Figure 2.4, the TR of edge-pair coverage criteria would be: {[n_0, n_1, n_2], [n_1, n_2, n_3], [n_0, n_1, n_4], [n_2, n_3, n_1], [n_3, n_1, n_4]} while a valid test path which covers this set of test requirements could be the same path used before: [n_0, n_1, n_2, n_3, n_1, n_4].

To define the next criterion we need to first define the simple path concept. A simple path in a given CFG is a path from n_i to n_j where no node appears more than once, with the exception of first and last node that can be identical. Considering this, a prime path is a maximal length simple path, that is a path from n_i to n_j that is a simple path and does not appear as a proper subpath of any other simple path. This criterion is defined as follows: "Prime Path Coverage (PPC): TR contains each prime path in G" [Offutt, 2008, p. 35]. Considering the Figure 2.4 the TR for prime path coverage criteria would be: {[n_0, n_1, n_2, n_3], [n_0, n_1, n_4]} and valid test paths which cover these requirements could be: [n_0, n_1, n_2, n_3, n_1, n_4] and [n_0, n_1, n_4].

Another useful testing criterion, called Complete Round Trip, starts a test in some node and ends on the same node. It is used to focus on the loops of the code. This criterion is defined as follows: "Complete Round Trip Coverage (CRTC): TR contains all round trip paths for each reachable node in G." [Offutt, 2008, p. 36]. The TR for this criteria, considering the Figure 2.4 would be: {n_1, n_2, n_3, n_1} and a valid test path that covers these test requirements is: [n_0, n_1, n_2, n_3, n_1, n_4].

2.2.2 Data Flow Coverage Criteria

After we have the definitions for a CFG already explained we can discuss the data flow criteria of graph coverage for source code. The premise to apply these criteria is to know the concepts of definition (*def)* and usage (*use)* of variables. A *def* is location where the program defines a value for a certain variable while the *use* is a location where this variable value is accessed by the program.

Finally knowing the concepts of *def* and *use* we will now discourse about *du-path.* A *du-pair with respect to a variable x* is a tuple with one *def* and one *use* of the same variable *x*, in which the *use* is reachable by the *def* following the graph flow. The path between the two nodes of the du-pair is said *def-clear* if no other *def* of *x* is found in the path. Consequently a *du-path* with respect to variable x is a def-clear path to reach the *use* of a *du-pair* starting at its *def* and contains all the nodes from this path, including the *def* and the *use* nodes.

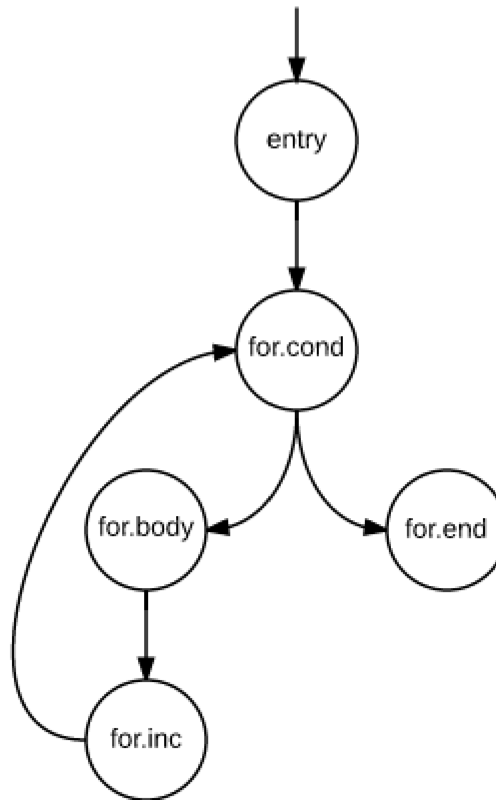
Figure 2.5 - fnc_move_data32 for data flow example.

```
void fnc_move_data32(uint8_t* destination_add, uint8_t* source_add){
        for(int i=0;i<4;i++)
                *destination_add++= (*source_add++)+0x30;
}
```

Source: Electronic Energy Meter Program

The control flow graph extracted from the source code of Figure 1.5 can be seen in Figure 2.6 and will be used as running example for the next sections and chapters.

Figure 2.6 fnc_move_data32 CFG



Source: The Author

Table 2.1 assumes that we create a CFG from the source code above and list all *uses* and *defs* for each node consisting of a basic block.

Table 2.1 - Defs and uses at each node in the CFG for fnc_move_data32.

Node	def	use
entry	{i}	
for.cond		{i}
for.body		
for.inc	{i}	{i}
for.end		

Source: Introduction to Software Testing (2008, p. 56)

Considering the criteria defined until now on sections 2.2.1 and 2.2.2 and the CFG of Figure 2.6, Table 2.2 displays a complete example of test requirements and test paths that satisfy each criterion accordingly. The first column lists the criterion, the second column lists the correspondent test requirements and on third column we have the test paths that satisfy the criterion, that is, cover all requirements defined.

Table 2.2 - Coverage criteria example.

Criteria	Test Requirement	Test Path
Node Coverage	{entry, for.cond, for.body, for.inc, for.end}	[entry,for.cond,for.body,for.inc,for.cond, for.end]
Edge Coverage	{(entry,for.cond),(for.cond,for.body),(for.cond,for.end),(for.body,for.inc),(for.inc,for.cond)}	[entry,for.cond,for.body,for.inc,for.cond, for.end]
Edge Pair Coverage	{(entry,for.cond,for.body),(for.cond,for.body, for.inc),(for.body,for.inc, for.cond),(for.inc,for.cond, for.body)}	[entry,for.cond,for.body,for.inc,for.cond,for.end]
Prime Path	{(entry,for.cond,for.body,for.inc,for.cond),(entry,for.cond,for.end)}	[(entry,for.cond,for.body,for.inc,for.cond),(entry,for.cond,for.end)]
Complete Round Trip	{for.cond,for.body,for.inc, for.cond}	[entry,for.cond,for.body,for.inc,for.cond,for.end]
Du-Path	{(entry,for.cond),(entry,for.cond,for.inc)}	[entry,for.cond,for.end][entry,for.cond,for.inc,for.cond,for.end]

Source: The Author

2.3 LLVM

LLVM or Low Level Virtual Machine *"[…] is a collection of modular and reusable compiler and toolchain technologies."* [LLVM.org, [200-?]]

The feature offered by LLVM that is needed for the approach proposed in this study is the compiler infrastructure developed to optimize programs in compilation time. Such feature is able to get the intermediate representation generated by the compiler and improve it, returning another optimized intermediate file. This intermediate file LLVM

creates is a CFG representation of the source code compiled and will be used to apply the coverage criteria techniques anteriorly introduced.

2.3.1 LLVM Intermediate Representation

The intermediate representation generated by LLVM is a low-level programming language similar to assembly. It is composed of modules that consist of functions, global variables, and symbol table entries of the input program.

On this representation each method starts with a "*define*" keyword and contains a list of basic blocks that consists in a CFG of the source code. A label is assigned to each one of the basic blocks and it will represent the identification of each node of the CFG. On the first line of the basic block we can also find its predecessors, with the exception of the first basic block, which is a special case. The first basic block, or first node, is the entrance point of the method and does not have any predecessor.

Figure 2.1 is a representation of the intermediate code generated by LLVM from the method fnc_move_data32 presented on Figure 1.5. This code is a CFG composed by a single node identified as "entry". The "define" reserved word represents a new method or the equivalent to a new graph and the variables are the names preceded by a "%" symbol in the intermediate code.

Figure 2.7 - LLVM Intermediate code of fnc_move_data32.

```
define void @_Z15fnc_move_data32PhS_(i8* %destination_add, i8* %source_add) nounwind uwtable {
entry:
  %destination_add.addr = alloca i8*, align 8
  %source_add.addr = alloca i8*, align 8
  %i = alloca i32, align 4
  store i8* %destination_add, i8** %destination_add.addr, align 8
  store i8* %source_add, i8** %source_add.addr, align 8
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32* %i, align 4
  %cmp = icmp slt i32 %0, 4
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i8** %source_add.addr, align 8
  %incdec.ptr = getelementptr inbounds i8* %1, i32 1
  store i8* %incdec.ptr, i8** %source_add.addr, align 8
  %2 = load i8* %1, align 1
  %conv = zext i8 %2 to i32
  %add = add nsw i32 %conv, 48
  %conv1 = trunc i32 %add to i8
  %3 = load i8** %destination_add.addr, align 8
  %incdec.ptr2 = getelementptr inbounds i8* %3, i32 1
  store i8* %incdec.ptr2, i8** %destination_add.addr, align 8
  store i8 %conv1, i8* %3, align 1
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %4 = load i32* %i, align 4
  %inc = add nsw i32 %4, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret void
}
```

Source: The Author

The intermediate code (as presented in Figure 2.7) is the input to the tool proposed in this work. From this code, a CFG is built and different test requirements sets can be generated according to different graph-based coverage criteria.

3 IMPLEMENTATION

In this chapter we will discuss the approach used to create a tool capable of applying the coverage criteria presented before.

As discussed on the previous chapter, CFGs are the most common artifact used to represent source codes and it will be the base of the test coverage criteria tool developed for this work.

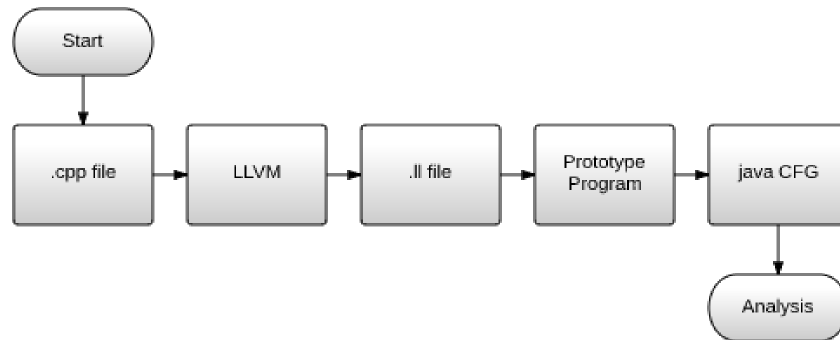
3.1 The Prototype Tool

The algorithms developed to apply graph-based coverage criteria presented on Chapter 1 is based on LLVM output. Figure 3.1 gives an overview of the usage flow. First the original program should be compiled with LLVM, generating the intermediate file that represents the CFG of the source code. Once we have the representation of a CFG of each method, the tool builds a java representation of these CFGs. Having these artifacts of the source code we can now use the testing techniques proposed. These techniques will generate the TR equivalent to each criterion selected, according to a given graph and its attributes as nodes, edges and variables.

3.1.1 Input Parsing

The code coverage application receives as input a file with all methods of a class, each one of these methods will be interpreted as a distinct graph. This graph will be composed by nodes, each node has an identification and has the information of predecessor nodes, the code of the basic block and variables defined and used inside of it.

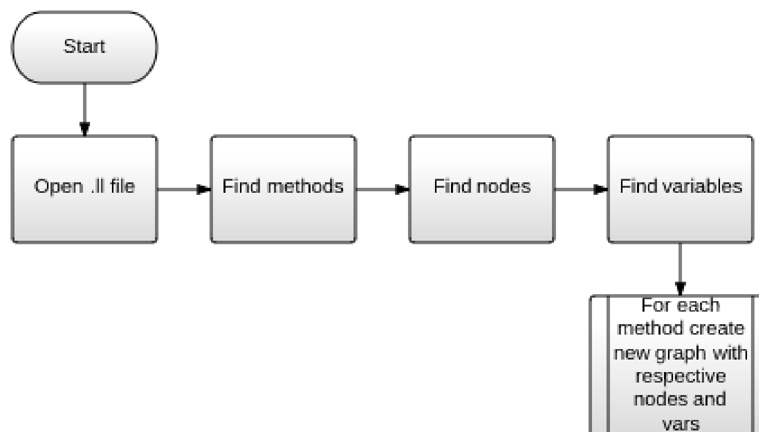
Figure 3.1 -File Lifecycle Flowchart



Source: The Author

Figure 3.2 shows how the application handles the input file and its information converting the intermediate file (.ll) created by LLVM in graph structures. The graphs are first identified, getting their names, then the nodes are created and their corresponding codes are stored. After all the nodes are read from the input file, they are matched with their respective graphs and saved as its attributes. The subprocess of creating a new graph is presented next.

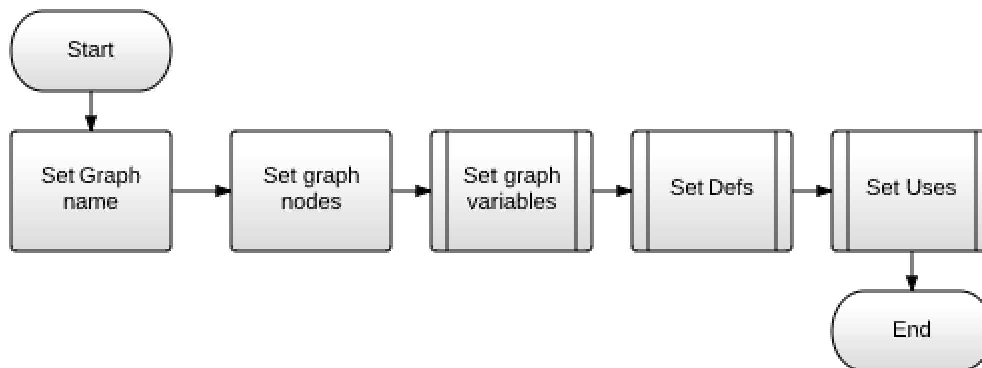
Figure 3.2 - Parsing Intermediate File Flowchart



Source: The author

In the subprocess of creating a new graph we set all the relevant information of the graph and its nodes. The graph receives a name which is the same of the method, its nodes are set, all the valid variables of the graph are stored and the sets of uses and definitions are created for each node in the graph. This basic flow is represented in the Figure 3.3, we can notice it is composed by other subprocessess.

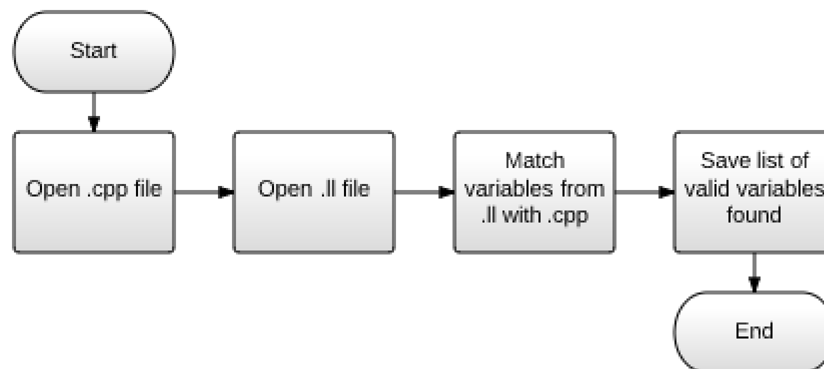
Figure 3.3 - Graph Creation Flowchart



Source: The author

The first subprocess we find on the flow of Figure 3.3 is for setting all the graph variables. This information is stored with the graph because LLVM intermediate file includes temporary variables that are not in the original source code. This subprocess defines therefore which variables are really important for the CFG. The process that sets these data works by reading the intermediate and the original file and matching the variables. Variables that are in both files are considered valid variables and are stored in the graph. We can analyze how the tool executes this feature by seeing the Figure 3.4.

Figure 3.4 - Set graph variables flowchart

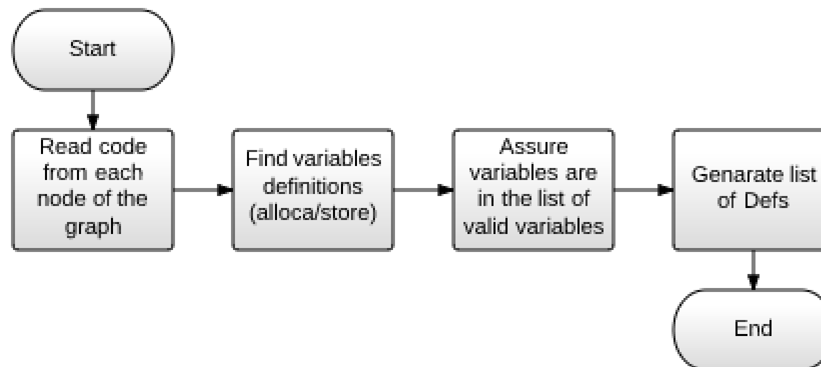


Source: The author

The next subprocess, described on Figure 3.5, is also related to the variables and is responsible for storing all its definitions points. The variable definition is not a graph information anymore, but a node attribute and will be set for each node in the graph. As we can see in the workflow, the process starts by getting each one of the nodes, and reading from its code which are the variables defined for each line of code inside of this basic block. After checking it is a valid variable (it belongs to the list of variables in the graph) the variable can be added to the list of *defs* of the node.

The first step to locate a *def* of a variable is to find the reserved words “alloca” or “store”. Once we have a line containing one of these words, we get the variable being defined by finding the symbol “%” in the beginning of the string which identifies a variable. After checking it is a valid variable, that is, it is stored by the graph as a variable, it is added to a list of *defs* which will be later assigned to the node.

Figure 3.5 - Set variables definitions flowchart

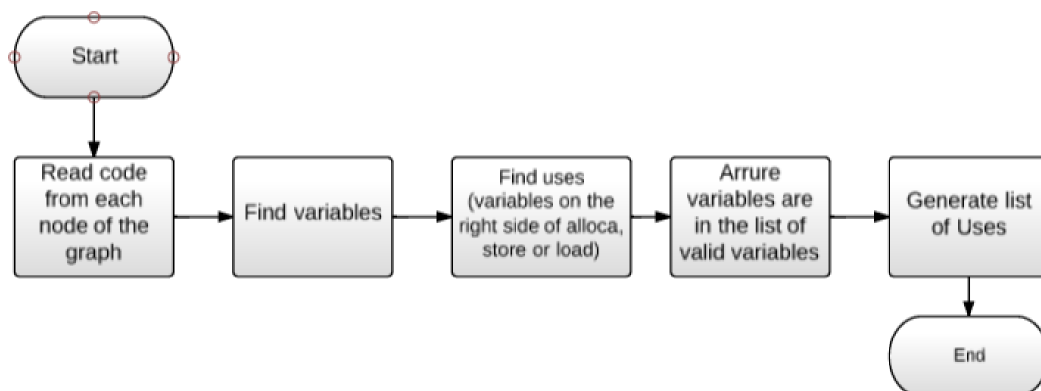


Source: The author

The subprocess of setting the uses works similarly to the "*set defs*" just presented. As we can see on Figure 3.6, for each node of the graph, all their lines of code are checked to find a use of a variable. Whenever a use is found it is added to a list that will be assigned to the node in the end of the process of validation of the whole basic block.

A use can be identified in the intermediate file by finding a variable (string starts with "%"). If there is no "load", "alloca" or "store" in the line it is automatically added to the list of *uses*, as long as it is a valid variable, and not a temporary one. In case of finding the reserved words that identify a *def*, the use will still be searched on the other part of the line, since the *def* of a variable could mean a *use* of another one.

Figure 3.6 - Set variables uses flowchart



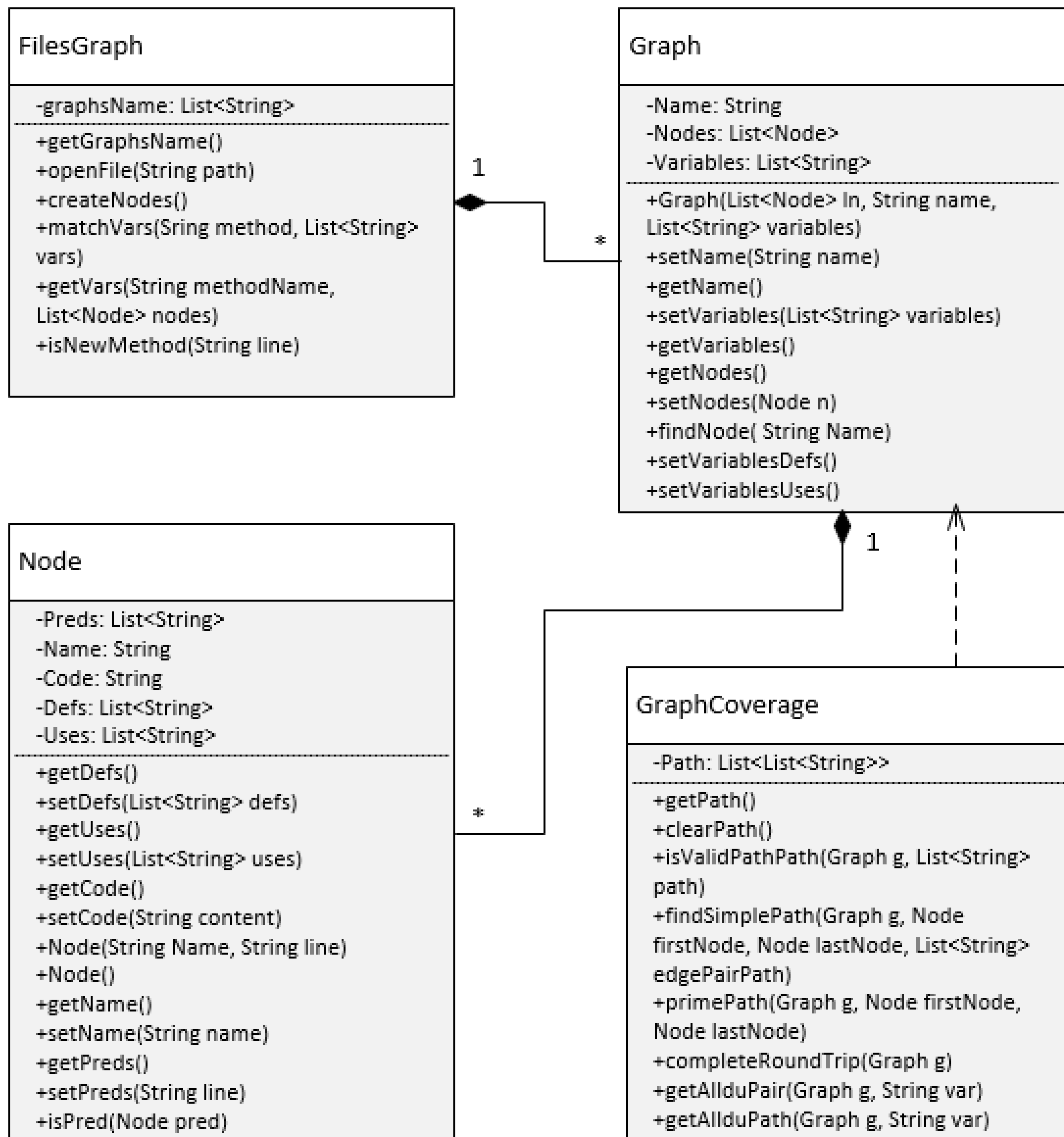
Source: The author

3.2 Structure of the Proposed Approach

Now we know how the application interprets the intermediate file generated by LLVM to get the CFGs necessary for source code analysis, let us analyze how the proposed tool is structured. In Figure 3.7 we can see the class diagram representation of the code coverage tool.

The program is separated in four classes, besides the main and the UI. The “FilesGraph” is responsible for taking the information of the file and building the graph from the data found. The “Graph” class represents the CFG itself and stores its name, a list of its nodes and a list with all the valid variables contained in the graph. The “Node” class includes the name (that is the identification of the basic block), a list with all names of the predecessor nodes, the code of the basic block and defs and uses of variables. Finally we have the “GaphCoverage” class that consists in all methods responsible for returning the test requirements and applying the code coverage criteria that are supposed to be implemented on future releases.

Figure 3.7 - Application Class Diagram



Source: The author

3.3 Implementation of the Coverage Criteria

On this section we will discuss about the coverage criteria selected to be implemented and the methods needed to cover the CGF the application receives as input.

Graph-based coverage criteria analysis starts by specifying a set of Test Requirements, TR. The developed tool is able generate all the TRs for the input graphs considering 9 different coverage criteria: Prime Path coverage, Complete Round Trip coverage, All Du-pair coverage, All Du-path coverage, Du-path coverage, Def-path coverage, Use-path coverage, All Def-path coverage and finally All Use-path coverage. Each method implemented toward this end is explained next. Other criteria as Node coverage and Edge coverage were omitted from this work for being trivially extended from developed methods.

3.3.1 Find Simple Path

The recursive method “findSimplePath” shown in Figure 3.8, receives as input a graph, two nodes and a path that is a list of nodes. The path starts empty and it is incremented with visited nodes on each step of the recursion executed. Each node reached in the recursion is added to the path. When a final node is found without reaching the target it is removed from the path and a new path is searched from the last node in the list of nodes. When the target is reached it adds the path to a global variable which stores the valid paths found. Then the last node is removed (the target) and the search for valid paths continues from the last node in the path variable.

Essentially this method is used to validate whether there is a valid simple path between two nodes. This functionality is used by other methods to validate links between two given nodes.

Figure 3.8 - Find Simple Path method

```
public void findSimplePath(Graph g, Node firstNode, Node lastNode, List<String> simplePath)//call clearPath() bef
{
    if( (!simplePath.contains(lastNode.getName()))  || ( //not a cycle
        (simplePath.contains(lastNode.getName()) && (lastNode.equals(firstNode)) ))){

        simplePath.add(lastNode.getName());

        if(lastNode.equals(firstNode))
        {
            if(simplePath.size()>1)
                addPath(new ArrayList<String>(simplePath));//add a valid path to results
            if(simplePath.size()>1)
            {
              simplePath.remove(simplePath.size()-1);//remove last node to find another path
              return;
            }
        }
        if(lastNode.getPreds().size()>0)//if there is still a predecessor for the node, build path with preds
        {
            for(String s : lastNode.getPreds())
            {
                findSimplePath(g, firstNode, g.findNode(s),simplePath);//test preds of node added
            }
        }
        simplePath.remove(simplePath.size()-1);//path not found, remove last node added
    }
}
```

Source: The author

3.3.2 Prime Path

Before defining the prime path we need to have another definition explained, the simple path. A path from node n_i to node n_j where no node is visited more than once, with the exception of the first node that could be the same as the last one, is called a simple path. That is, simple paths do not contain loops, but it can be a loop.

A prime path is a simple path that does not appear as a proper subpath of other simple paths.

To find out all prime paths in a graph between two nodes, this method uses the "findSimplePath" method to find the paths between a pair of nodes in the graph. After generating all simple paths, it selects the ones that are not contained in the others.

To have the prime path criteria completed covered it would be necessary to run this method for each pair of nodes existent in the graph.

Figure 3.9 - Prime Path method

```
public void primePath(Graph g, Node firstNode, Node lastNode)
{
    List<List<String>> primePath = new ArrayList<List<String>>();
    boolean isPrimePath;

    clearPath();
    edgePair(g,firstNode,lastNode, new ArrayList<String>());//get all possible paths

    for(List<String> ls : getPath())//for all paths available test if it's contained in another path
    {
        isPrimePath = true;
        for(List<String> lsnext : getPath())
        {
            if((lsnext.containsAll(ls)) && (!lsnext.equals(ls)) )//lsnext contains ls, so ls is not a prime path
            {
                isPrimePath = false;
            }
        }
        if(isPrimePath)
            primePath.add(ls);//none of other paths contains ls, so add ls to the prime path list
    }

    clearPath();
    Path.addAll(primePath);
}
```

Source: The author

3.3.3 Is Valid Path

The “isValidPath” method (Figure 3.10) receives a path as input and test if all the nodes of this path are reachable. As the nodes store their predecessors it takes the last node of the path and validate if the predecessor node in the path is also a predecessor in the graph. If the first node, that is the final target, is reached visiting all the nodes in the path the method returns a confirmation of the validity of the path tested.

This method will be useful to validate input test paths when coverage features are developed.

Figure 3.10 - Is Valid Path method

```
public boolean isValidPath(Graph g, List<String> path)
{
    //get last node from path entered
    for(int i = path.size()-1; i>=1 ;i--)
    {   //test predecessor

        if(!g.findNode(path.get(i)).isPred(g.findNode(path.get(i-1))))
            return false;
    }

    return true;
}
```

Source: The author

3.3.4 Complete Round Trip

A round trip path is a prime path of length greater than 0, that starts and ends at the same node. As we can notice the round trip is a variation of a prime path, the difference is it tests the loops for a given node. Taking advantage of this concept this method uses the prime path method implemented but uses as input the same node as initial and end point of the path.

Figure 3.11 - Complete Round Trip method

```
public void completeRoundTrip(Graph g, Node n)
{
    clearPath();
    primePath(g, n, n);
}
```

Source: The author

3.3.5 All DuPairs

A du-pair exercises the data flow criteria since it focuses on testing the correct use of variables in the program. This method returns all pairs of definition and uses found on a graph for a certain variable.

First the “allDuPair” method looks for uses of a variable *v.* When a use is found it looks for definitions of *v* and uses the “findSimplePath” for each pair of def and use found to test whether they are connected. Finally if there is a path between both def and use nodes of v, this method tests if this is a def-clear path. If this is the case, the nodes are stored in a list of valid du-pairs found on the graph considering the variable entered as parameter.

Figure 3.12 - All Du-Pair method

```
public List<String> allDuPair(Graph g, String var)
{
    List<String> DuPairs = new ArrayList<String>();
    for(Node n : g.getNodes())
    {
        if(n.getUses().contains(var))//for all nodes with use for var
        {
            for(Node n2 : g.getNodes())
            {
                clearPath();
                if((!n.getName().equals(n2.getName())) && (n2.getDefs().contains(var)))//for all nodes with a def i
                {
                    findSimplePath(g,n2,n, new ArrayList<String>());//test if there is a path between them
                    if(!getPath().isEmpty())
                    {
                        for(List<String> path : getPath())
                        {
                            if(isDefClear(g, path, var))//if a path is found and it's defclear it's a du-pair
                            {
                                DuPairs.add("Def: " + n2.getName() + " Use: "+ n.getName());
                            }
                        }
                    }
                }
            }
        }
    }
    return DuPairs;
}
```

Source: The author

3.3.6 DuPath

The du-path method works exactly as the du-pair one but now focusing on the whole path while the du-pair considers just the nodes. It stores the entire path between the def and use of a variable *v.*

Figure 3.13 - All Du-Path method

```
public List<List<String>> allDuPath(Graph g, String var)//return all duPaths of all du-pairs
{
    List<List<String>> AllDuPath = new ArrayList<List<String>>();
    for(Node n : g.getNodes())
    {
        if(n.getUses().contains(var))
        {
            for(Node n2 : g.getNodes())
            {
                clearPath();
                if((!n.getName().equals(n2.getName())) && (n2.getDefs().contains(var)))
                {
                    edgePair(g,n2,n, new ArrayList<String>());
                    if(!getPath().isEmpty())
                    {
                        for(List<String> path : getPath())
                        {
                            if(isDefClear(g, path, var))
                            {
                                AllDuPath.add(path);
                            }
                        }
                    }

                }
            }
        }
    }

    return AllDuPath;
}
```

Source: The author

3.3.7 All DefPath

The allDefPath method is also based on data flow criteria, and is based on the same concepts just discussed (du-pair and du-path), but it focuses on the variables definition.

The defPath method receives a node as parameter and checks whether this node has a definition for a variable v, also entered as input. In case the node has a definition of v, the method searches for a use of v. For the uses found, the method tests (using the "findSimplePath" method) if they are connected and if the path is def-clear. On the other hand the method to find all the defPath uses the method just described and applies for each node on the graph, as we can verify on Figure 3.14.

Figure 3.14 - Def-Path methods

```
public List<List<String>> allDefPath(Graph g, String var)
{
    List<List<String>> AllDefPath = new ArrayList<List<String>>();

    for(Node n : g.getNodes())
    {
        if(n.getDefs().contains(var))
        {
            List<List<String>> defPath = new ArrayList<List<String>>();
            defPath.addAll(defPath(g,n,var));
            if(!defPath.isEmpty())
                AllDefPath.addAll(defPath);
        }
    }

    return AllDefPath;
}
public List<List<String>> defPath(Graph g, Node n, String var)//for each use of var find a def of var
{
    List<List<String>> DefPath = new ArrayList<List<String>>();
    if(n.getDefs().contains(var))//input node has a var definition
    {
        for(Node nUse : g.getNodes())//found the nodes of the graph with var use
        {                           //test if there is a path connecting them and its def-clear
            if(nUse.getUses().contains(var))
            {
                clearPath();
                findSimplePath(g,n,nUse,new ArrayList<String>());
                for(List<String> path : getPath())
                {
                    if(isDefClear(g,path,var))
                        DefPath.add(path);
                }
            }
        }
        if(DefPath.isEmpty())
            System.out.println("No use found for def: " + n.getName() + " " + var);
        return DefPath;
    }
    else
    {
        System.out.println(n.getName() + "doesn't contain a def of: " + var);
        return null;
    }
}
```

Source: The author

3.3.8 All UsePaths

The allUsePath method is also based on data flow criteria, but it focuses on the variables usages an return all the use paths considering all uses of a variable *v*.

The usePath method receives a node as parameter and validates this node has a use of a variable v*,* also entered as input. In case the node has a use of *v* it searches for a def of *v.* For the uses found, the method tests using the "isValidPath" method if they are connected and it the path is def-clear. The use-path returns the paths containing all definition nodes corresponding to a given use of *v*.

Figure 3.15 - Use-Path methods

```
public List<List<String>> allUsePath(Graph g, String var)
{
    List<List<String>> AllUsePath = new ArrayList<List<String>>();

    for(Node n : g.getNodes())
    {
        if(n.getUses().contains(var))
        {
            List<List<String>> usePath = new ArrayList<List<String>>();
            usePath.addAll(usePath(g,n,var));
            if(!usePath.isEmpty())
                AllUsePath.addAll(usePath);
        }
    }
    return AllUsePath;
}
public List<List<String>> usePath(Graph g, Node n, String var)
{
    List<List<String>> UsePath = new ArrayList<List<String>>();
    if(n.getUses().contains(var))//input node has a var use
    {
        for(Node nDef : g.getNodes())//found the nodes of the graph with var def
        {                            //test if there is a path connecting them and its def-clear
            if(nDef.getDefs().contains(var))
            {
                clearPath();
                findSimplePath(g,nDef,n,new ArrayList<String>());
                for(List<String> path : getPath())
                {
                    if(isDefClear(g,path,var))
                        UsePath.add(path);
                }
            }
        }
        if(UsePath.isEmpty())
            System.out.println("No def found for use: " + n.getName() + " " + var);
        return UsePath;
    }
    else
    {
        System.out.println(n.getName() + "doesn't contain a use of: " + var);
        return null;
    }
}
```

Source: The author

3.3.9 DefClear

The def-clear concept was much explored until now in the methods based on data flow criteria. The condition to say a du-path is def-clear is that no other definition of a variable v appears between the entry point of the path, which is a definition of variable *v,* and the final node of the path, which is a use of *v*.

To get this condition satisfied this method receives as input the graph, the path to be tested and the variable in question. For each node in the given path a definition of *v* is searched. , if it reaches the final node without finding other definition for v it returns the value true, asserting the given path is in fact def-clear.

Figure 3.16 - Is Def-Clear method

```
public boolean isDefClear(Graph g, List<String> path, String var) {
    for(String n : path)
    {
        if((!n.equals(path.get(0))) && (!n.equals(path.get(path.size()-1))))
        {
            if(g.findNode(n).getDefs().contains(var))
            {
                return false;
            }
        }
    }
    return true;
}
```

Source: The author

3.4 Performance Analysis

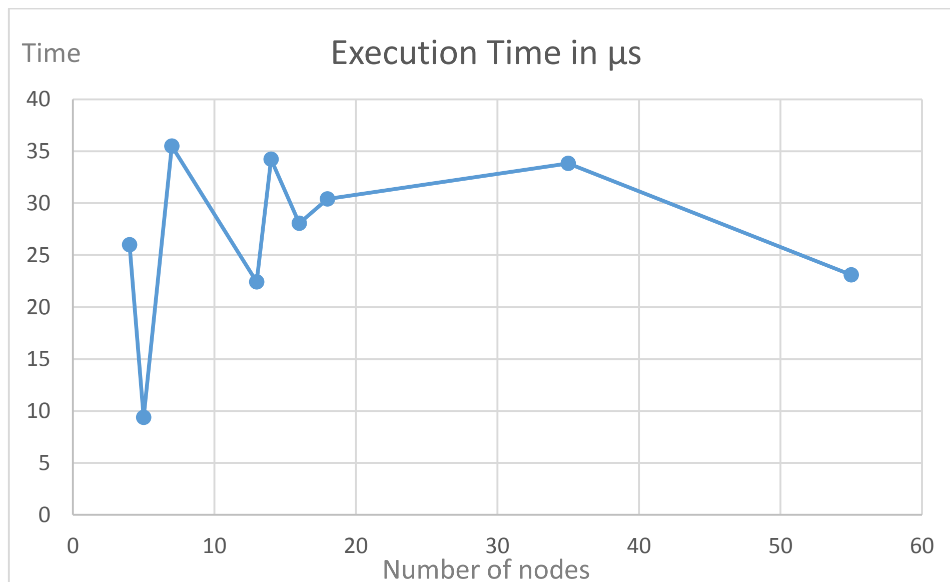
The algorithms developed to generate the Test Requirements (TR) for all criteria considered by this prototype were built based on the definitions presented by Offutt (2008) and considering the input which would be received by LLVM.

The average complexity of the algorithms implemented are directly proportional to the cyclomatic configuration of the graphs. As the propose was to work with object oriented code, the optimization of the algorithms were not considered to be a concern as this type of programming structure splits the logic into small methods which will result on graphs with reduced number of nodes.

Figure 3.17 represents the average time spent to run the method “findSimplePath” considering increasing number of nodes. The data was generated from the average of several executions of the method considering the same graph and changing the pair of

nodes used as parameters and also calculating the time for other graphs with same number of nodes but different number of edges.

Figure 3.17 - “findSimplePath” average execution time



Source: The Author

From the graph of Figure 3.17 we can verify the time average for the execution of “findSimplePath” method not increased considerably for any of the of the graphs from the real program tested, which means scalability did not show to be a concern considering the context of object oriented code.

4 EXPERIMENTAL RESULTS

This chapter focuses on showing the application of the proposed tool to an intermediate code created by LLVM for the example of Figure 2.5. We will discuss how the input file was processed and tested using the methods described on Chapter 3.

4.1 Running Example

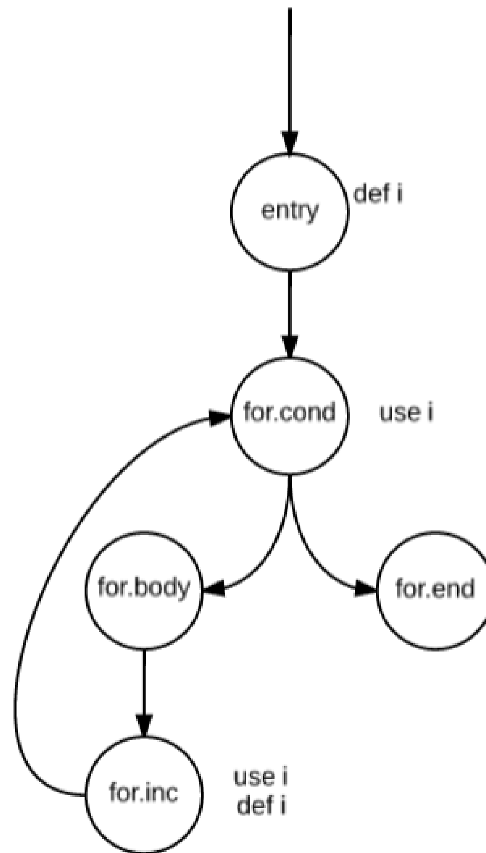
To be able to use the program that will run the coverage methods over the code, it is necessary to first have processed the code using LLVM. For the purpose of this work that is intended to cover Object Oriented codes and respecting LLVM current capabilities, we used a source code implemented in C++ .

Both files containing the code, original and intermediate (.ll extension), should be placed in the root project folder in order to get required information for the tool and make it possible to perform the desired analysis.

4.1.1 Input Program

A control flow graph is an oriented graph built based on the flow of a program execution. The nodes of this graph represent basic blocks of code while the edges represent possible transfer of control flow from one basic block to another. The CFG can be extracted from the intermediate file required. For this specific running example the intermediate code created by LLVM has the following Control Flow Graph:

Figure 4.1 - CFG of input program with defs and uses



Source: The author

4.2 File Processing

The data structure needed by the coverage methods is achieved by a series of operations in the beginning of the process. Each method of the read file is transformed in a different graph which stores the nodes, its predecessors, its code and all the variables definition and uses of the basic blocks.

4.3 Test Execution

Coverage criteria is used to define a set of tests and inputs that should be used to cover a program. The choice of this specific set makes possible to decrease the cost of testing and still ensure the quality of the program, since it will be focusing on the critical points of the code.

For the running example used here the methods for generating the Test Requirements for coverage criteria chosen were the primePath(), which returns the test requirements of the prime path coverage criterion defined on chapter 2, and duPath(), which returns test requirements for the AllDefs coverage criterion for a given variable. Both methods will be better explained in the following sections. The input data, such as the path or the variable to be tested, on the other hand were chosen arbitrarily.

4.3.1 Prime Path

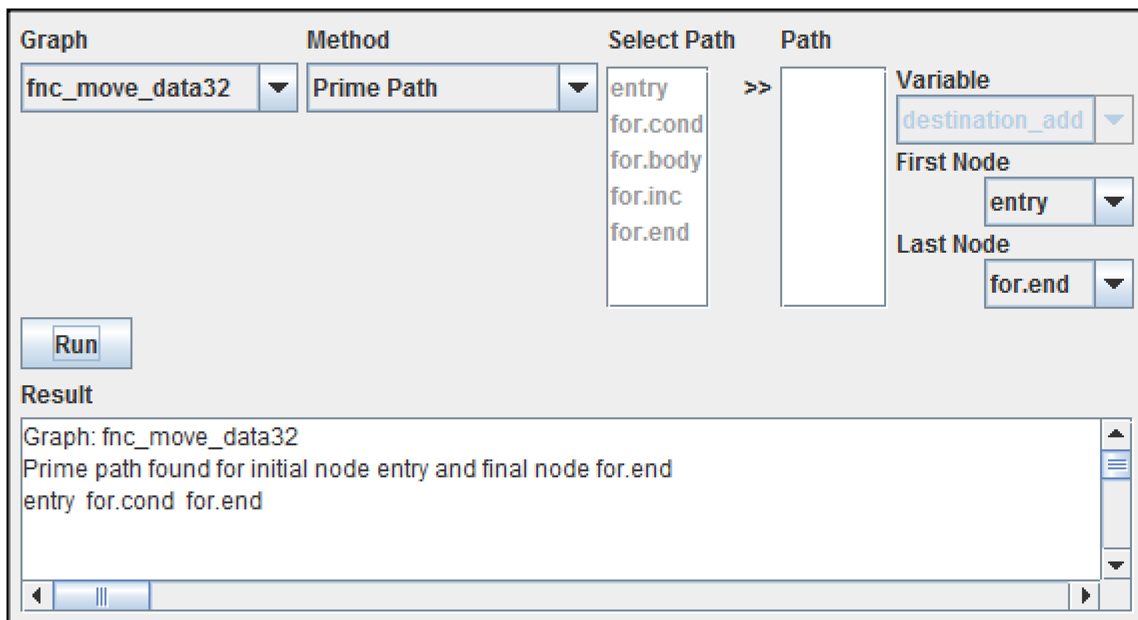
Prime path criterion is much explored for being simple and keep the number of test requirements down. It is a simple path of maximal length between two nodes, that is, the prime path does not appear as proper subpath of any other simple path.

The “primePath” method developed receives as parameter a graph and two nodes of this graph. All simple paths between the given nodes will be found by calling the method “findSimplePath”. Once we have all possible simple paths for the pair of nodes the method removes those that are subpaths of another. The Test Requirements (TR) returned will be the paths that satisfy the condition of not being a subpath of any other simple path found.

4.3.1.1 Prime Path Criteria Test Run

Figure 3.2 is an example of the tool execution for the CFG generated by LLVM and used as input for this experience. On this case the method Prime Path is being applied over the graph used as example (Figure 4.1) that corresponds to the source code displayed on Figure 2.5. To run the Prime Path option over the complete CFG the user sets the initial node to “entry” and the final node to “for.end”.

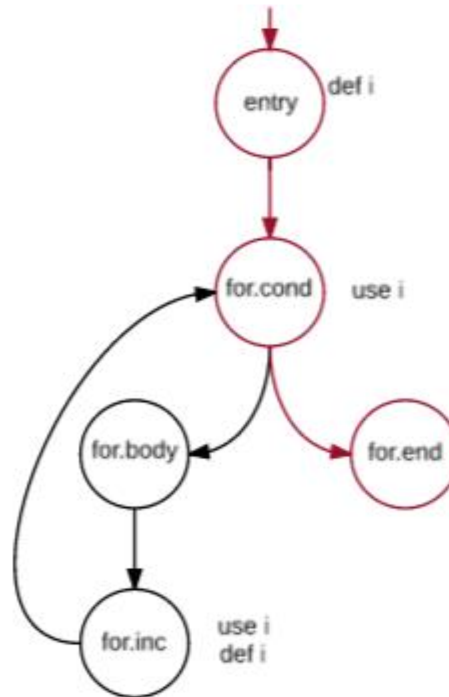
Figure 4.2 - Program execution for Prime Path criteria.



Source: The author

The result obtained can be verified in Figure 4.2. For this case one prime path is found starting on node “entry” and ending on node “for.end” and is represented by the following Test Requirement: {entry, for.cond, for.end}. The confirmation that the path is really contained in the graph and is a prime path can be seen in Figure 4.3.

Figure 4.3 - Prime path TR for “entry” to “for.end”.



Source: The author

4.3.2 Du-Path

The criterion discussed here is focused on validating the correct definition and use of variables. A definition of a variable is a location where it is created or has an assignment and a use of a variable happens when its value is accessed.

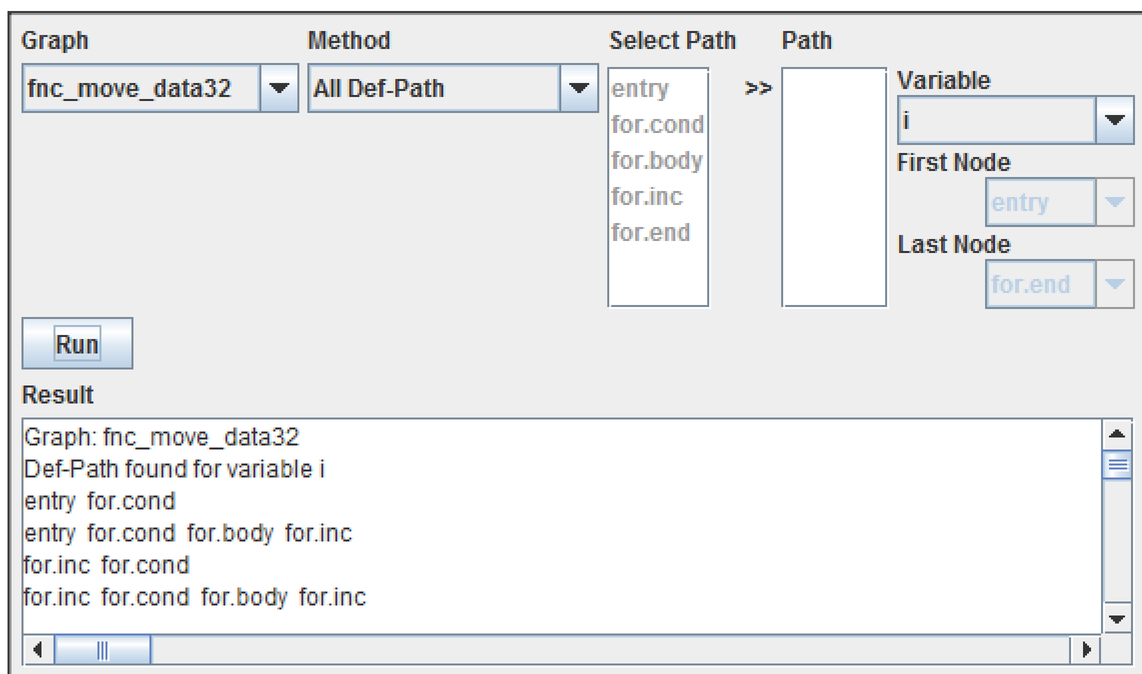
A du-path is a simple path with respect to a variable *v* that is def-clear and starts at a node where *v* is being defined and ends at a node where *v* is being used. Du-paths can be categorized in different groups, we will present here two of them: def-path and use-path.

4.3.2.1 All Def-Path Test Run

The def-path is the set of all du-paths with respect to a given definition. This means that given a node *n* where *n* contains a definition of a variable *v,* this method returns all du-paths starting at *n.*

Figure 4.4 is an example of the execution over the CFG provided as input for this experience and displayed on Figure 4.1. The method def-path is being applied over this graph, considering the variable *i.*

Figure 4.4 - Program call for def-path method



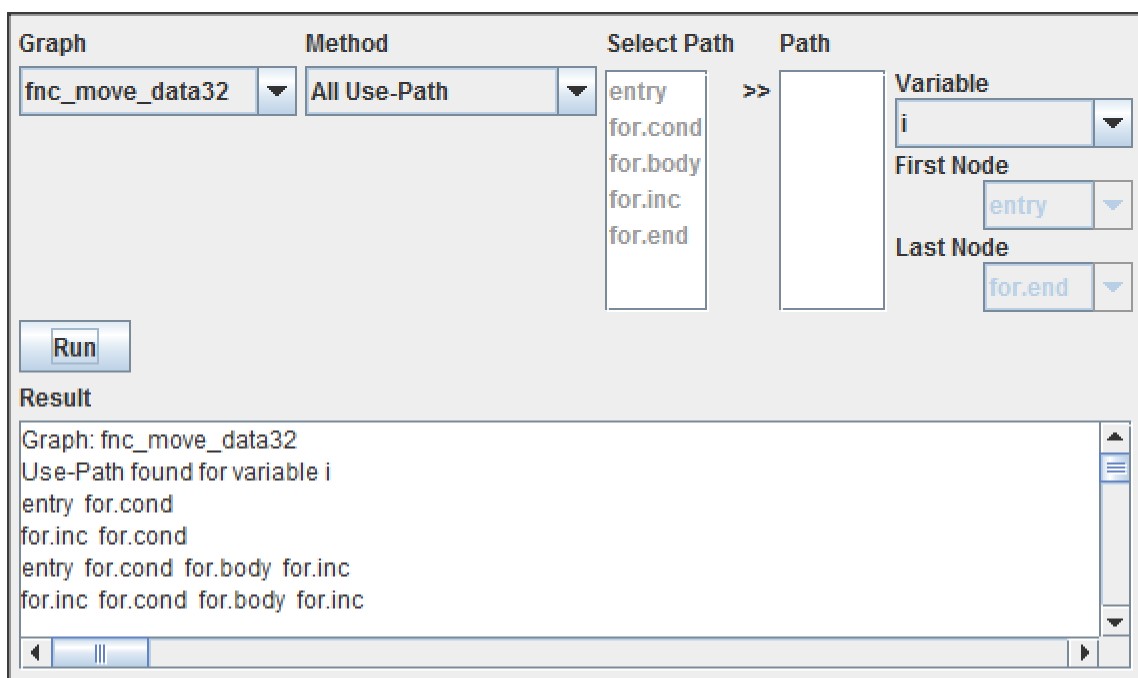
Source: The author

4.3.2.2 All Use-Path Test Run

The use-path is the set of all du-paths with respect to a given use. This means that, given a node *n* where *n* contains a use of a variable *v,* this method returns all du-paths ending at *n* and starting on a definition of *v* that can reach *n*.

In Figure 4.5 we can see the execution of the method All Use-Path considering the variable I and being applied over CFG of Figure 4.1.

Figure 4.5 - Program call for use-path method



Source: The author

On Figures 4.4 and 4.5 we can visualize the Test Requirements related to both criteria Def-Path and Use-Path. We can notice values returned are the same, since they are representing all def-clear paths for defs and uses of the same variable *i*. On Figure 3.6 we can validate the result found by the tool.

4.4 Validation in a Real System

For the validation of the proposed and developed prototype a set of 40 C++ files, part of a research of the Federal University of Rio Grande do Sul, were used as input to run the referenced tool.

The application tested consists on the software developed as part of an electronic energy meter.This software was developed as part of a research project and has only basic unit tests implemented.

4.4.1 Program Execution

On this section the results obtained running the prototype tool developed over the classes from the embedded software previously described will be presented. Considering the huge amount of data generated on this validation not all the classes will be displayed on next sections. The simpler examples were selected in order to be more explanatory and conclusive.

4.4.1.1 First Example

The first example is an execution of the prototype tool using as input file the intermediate code generated for class Calendar of the embedded software. The method chosen to be validated on the next steps was the “isNewMonth” for being easily representable.

Figure 4.6 - “isNewMonth” Source Code

```
bool Calendar::isNewMonth(){

    TDay daysOfTheMonth;

    if (_currentDateAndTime._monthOfTheYear == FEBRUARY && isLeapYear()){
        daysOfTheMonth = 29;
    }
    else {
        daysOfTheMonth = _numberOfDaysInTheMonth[_currentDateAndTime._monthOfTheYear];
    }

    return (_currentDateAndTime._dayOfTheMonth > daysOfTheMonth);
}
```

Source: Electronic Energy Meter Program

The correspondent intermediate code generated by LLVM from “isNewMonth” method can be visualized on Figure 4.7.

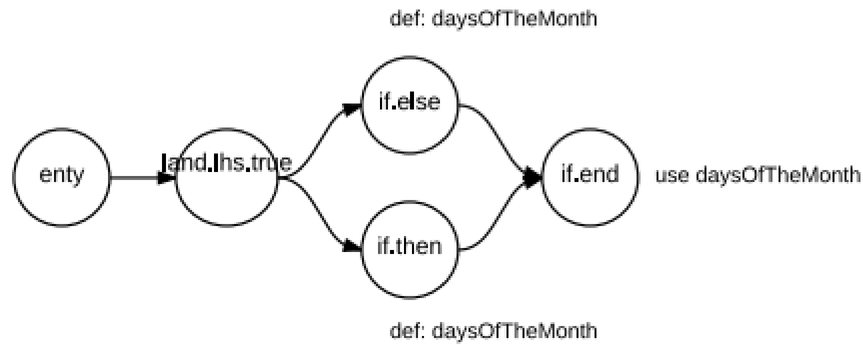
Figure 4.7 - “isNewMonth” Intermediate Code

```
define zeroext i1 @_ZN8Calendar10isNewMonthEv(%class.Calendar* %this) uwtable align 2 {
entry:
  %this.addr = alloca %class.Calendar*, align 8
  %daysOfTheMonth = alloca i8, align 1
  store %class.Calendar* %this, %class.Calendar** %this.addr, align 8
  %this1 = load %class.Calendar** %this.addr
  %_currentDateAndTime = getelementptr inbounds %class.Calendar* %this1, i32 0, i32 2
  %_monthOfTheYear = getelementptr inbounds %struct.structDateAndTime* %_currentDateAndTime, i32 0, i32 1
  %0 = load i32* %_monthOfTheYear, align 4
  %cmp = icmp eq i32 %0, 1
  br i1 %cmp, label %land.lhs.true, label %if.else

land.lhs.true:                                    ; preds = %entry
  %call = call zeroext i1 @_ZN8Calendar10isLeapYearEv(%class.Calendar* %this1)
  br i1 %call, label %if.then, label %if.else

if.then:                                          ; preds = %land.lhs.true
  store i8 29, i8* %daysOfTheMonth, align 1
  br label %if.end

if.else:                                          ; preds = %land.lhs.true, %entry
  %_currentDateAndTime2 = getelementptr inbounds %class.Calendar* %this1, i32 0, i32 2
  %_monthOfTheYear3 = getelementptr inbounds %struct.structDateAndTime* %_currentDateAndTime2, i32 0, i32 1
  %1 = load i32* %_monthOfTheYear3, align 4
  %conv = zext i32 %1 to i64
  %_numberOfDaysInTheMonth = getelementptr inbounds %class.Calendar* %this1, i32 0, i32 5
  %arrayidx = getelementptr inbounds [12 x i8]* %_numberOfDaysInTheMonth, i32 0, i64 %conv
  %2 = load i8* %arrayidx, align 1
  store i8 %2, i8* %daysOfTheMonth, align 1
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %_currentDateAndTime4 = getelementptr inbounds %class.Calendar* %this1, i32 0, i32 2
  %_dayOfTheMonth = getelementptr inbounds %struct.structDateAndTime* %_currentDateAndTime4, i32 0, i32 2
  %3 = load i8* %_dayOfTheMonth, align 1
  %conv5 = zext i8 %3 to i32
  %4 = load i8* %daysOfTheMonth, align 1
  %conv6 = zext i8 %4 to i32
  %cmp7 = icmp sgt i32 %conv5, %conv6
  ret i1 %cmp7
}
```

Source: Electronic Energy Meter Program

Considering the intermediate code of Figure 4.7 where we can see the nodes identified by their labels and their predecessors we obtained the CFG presented in Figure 4.8.

Figure 4.8 - “isNewMonth” correspondent CFG



Source: The Author

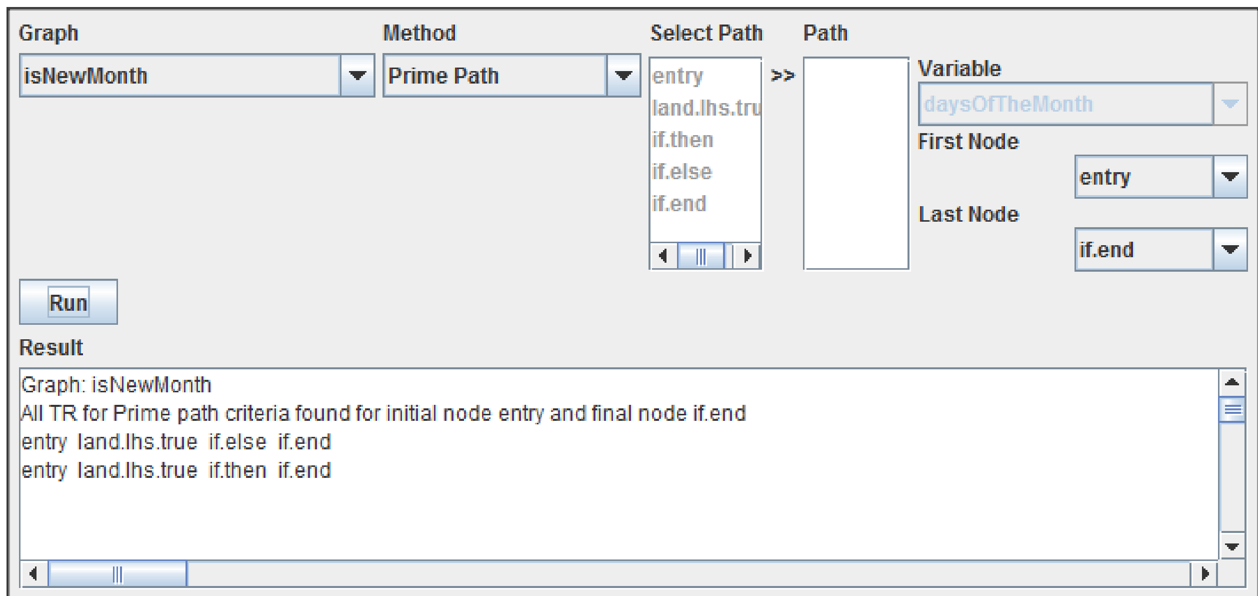
Using the data described on this section Figures 4.9 to 4.16 present the results obtained by the prototype for all the criteria available in the tool.

Figure 4.9 - Execution of Simple Path method for “isNewMonth“ graph and considering the nodes “entry” and “if.end”

Graph
isNewMonth
Method
Simple Path
Select Path
entry
land.lhs.tru
if.then
if.else
if.end
>>
Path
Variable
daysOfTheMonth
First Node
entry
Last Node
if.end
Run
Result
Simple Path(s) found for initial node: entry and final node: if.end
entry land.lhs.true if.else if.end
entry if.else if.end
entry land.lhs.true if.then if.end

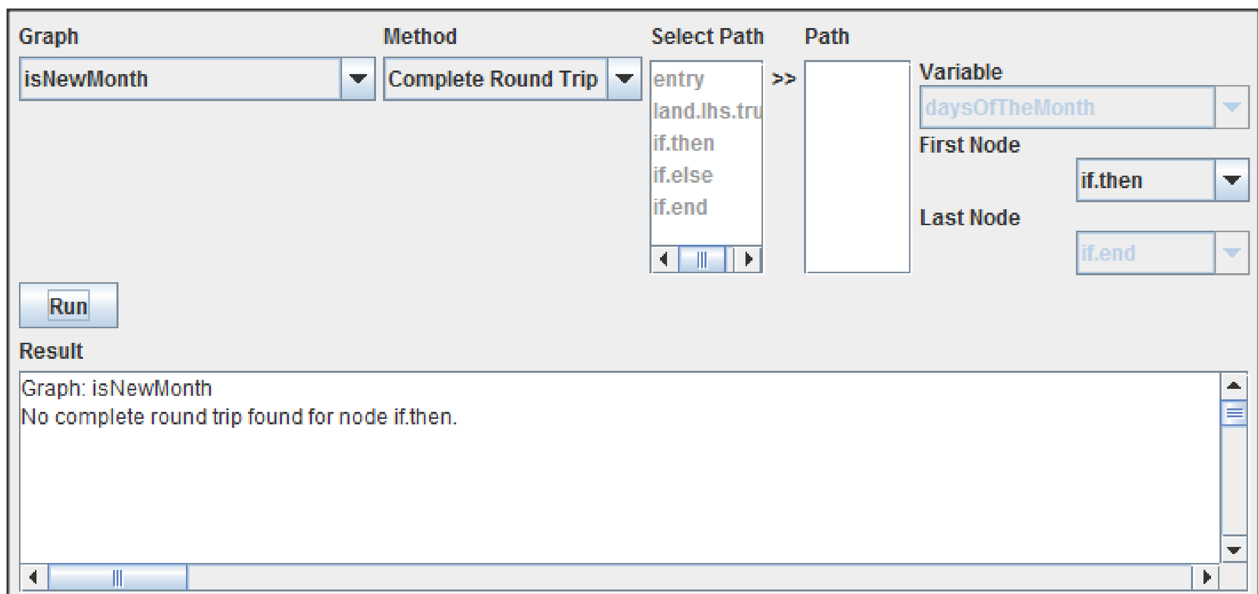
Source: The Author

Figure 4.10 - Execution of Prime Path method for “isNewMonth“ graph, considering the nodes “entry” and “if.end”



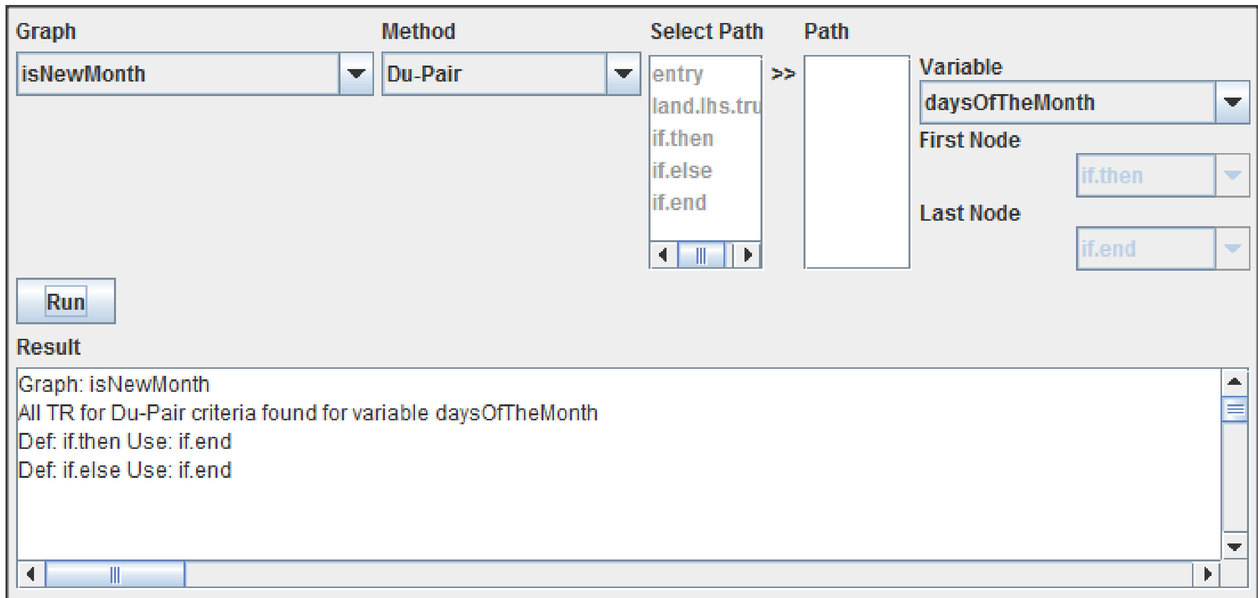
Source: The Author

Figure 4.11 - Execution of Complete Round Trip method for “isNewMonth“ graph, considering the node “if.end”



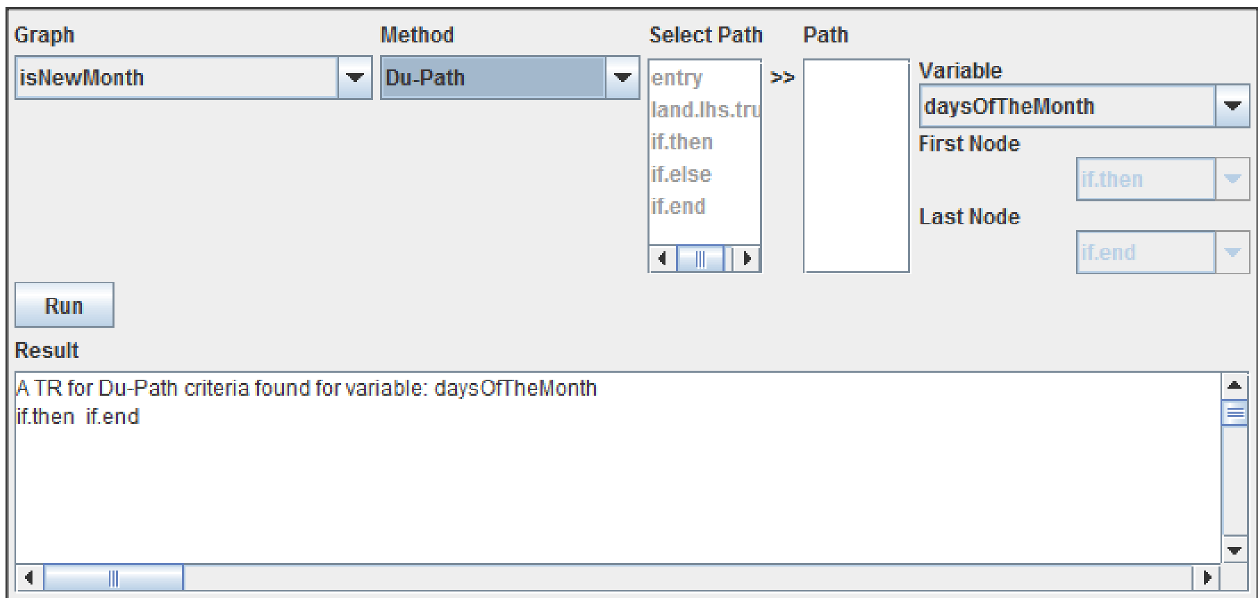
Source: The Author

Figure 4.12 - Execution of Du-Pair method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”



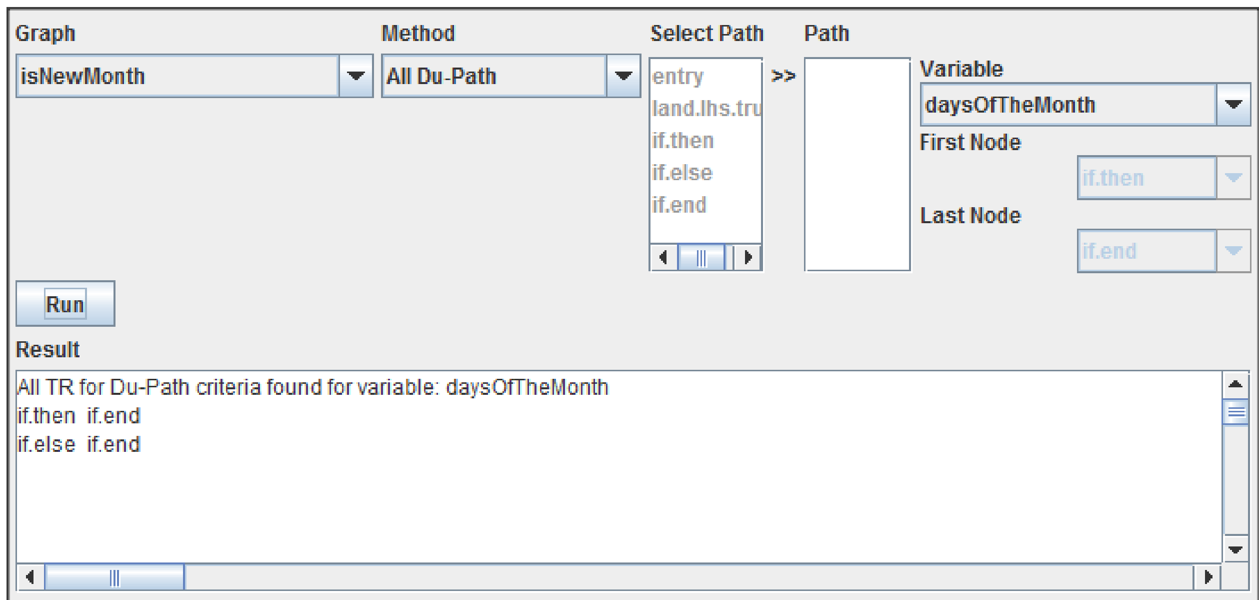
Source: The Author

Figure 4.13 - Execution of Du-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”



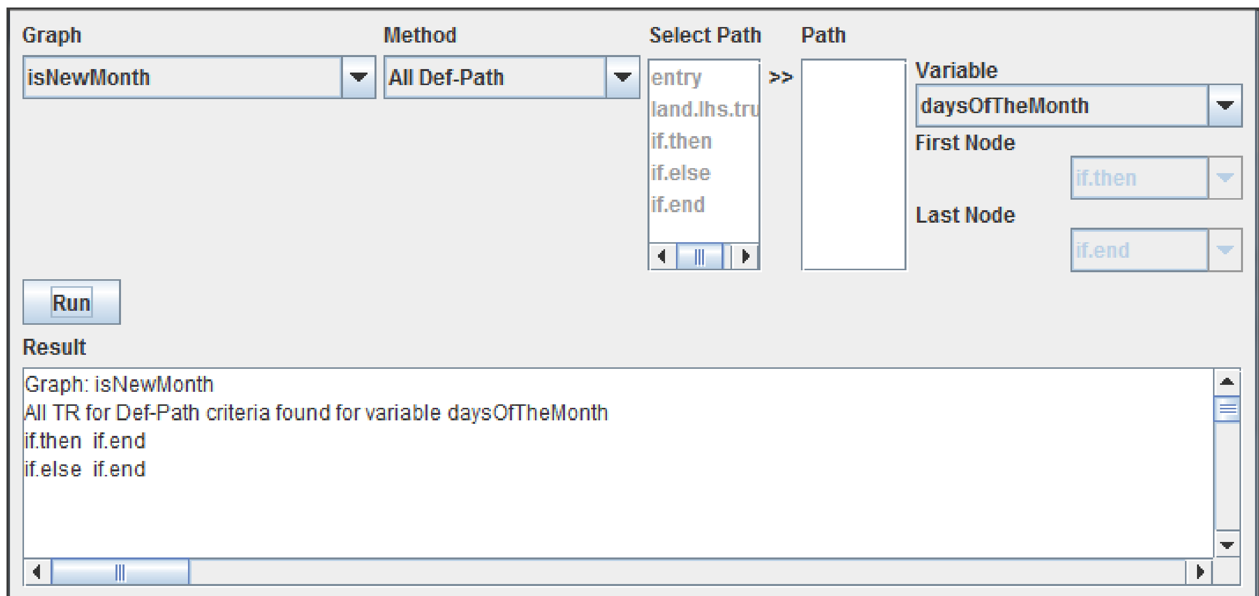
Source: The Author

Figure 4.14 - Execution of All Du-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”



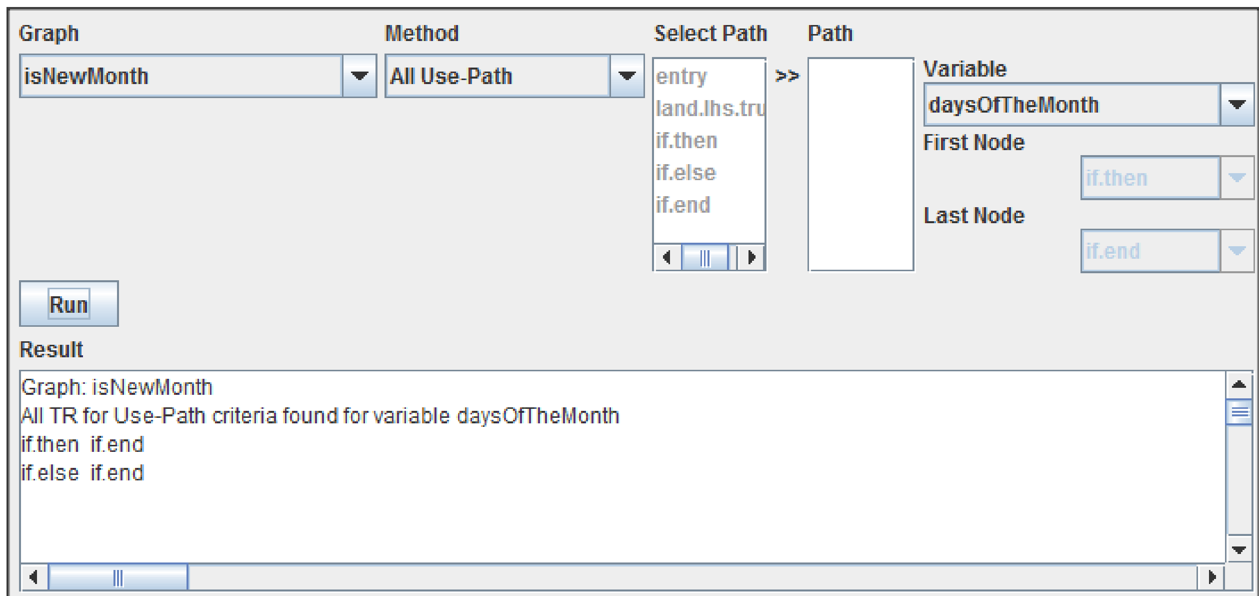
Source: The Author

Figure 4.15 - Execution of All Def-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”



Source: The Author

Figure 4.16 - Execution of All Use-Path method for “isNewMonth“ graph, considering the variable “daysOfTheMonth”



Source: The Author

4.4.1.2 Second Example

The second example is an execution of the prototype tool using as input file the intermediate code generated for class BillCommand of the embedded software. The method chosen to be validated on the next steps was the “update” for being easily representable.

Figure 4.17 - “isNewMonth” Source Code

```
void BillCommand::update(Subject* subject) {

    if (subject->getType() == INTERFACE_CONTROL){

        // InterfaceControl received notification from Bill button pushed
        cout << "Bill event is notified." << endl;
        // trigger a Bill Command
        setBillSignal();

    }
    else {
        throw subject;
    }

}
```

Source: Electronic Energy Meter Program

The corresponding intermediate code generated by LLVM from “update” method can be visualized on Figure 4.18 (part of the code was omitted since it was not essential for this purpose).

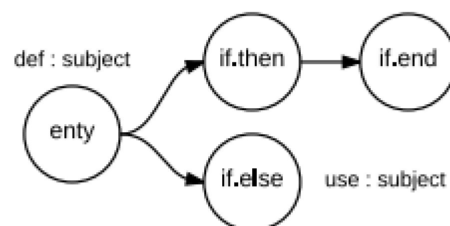
Figure 4.18 - “update” Intermediate Code

```
define void @_ZN11BillCommand6updateEP7Subject(%class.BillCommand* %this, %class.Subject* %subject)
unnamed_addr uwtable align 2 {
entry:
  %this.addr = alloca %class.BillCommand*, align 8
  %subject.addr = alloca %class.Subject*, align 8
  store %class.BillCommand* %this, %class.BillCommand** %this.addr, align 8
  store %class.Subject* %subject, %class.Subject** %subject.addr, align 8
  %this1 = load %class.BillCommand** %this.addr
  %0 = load %class.Subject** %subject.addr, align 8
  %call = call i32 @_ZN7Subject7getTypeEv(%class.Subject* %0)
  %cmp = icmp eq i32 %call, 3
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %call2 = call %"class.std::basic_ostream"* @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.s
  %call3 = call %"class.std::basic_ostream"* @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* %call2, %"class.std
  call void @_ZN11BillCommand13setBillSignalEv(%class.BillCommand* %this1)
  br label %if.end

if.else:                                          ; preds = %entry
  %exception = call i8* @__cxa_allocate_exception(i64 8) nounwind
  %1 = bitcast i8* %exception to %class.Subject**
  %2 = load %class.Subject** %subject.addr, align 8
  store %class.Subject* %2, %class.Subject** %1
  call void @__cxa_throw(i8* %exception, i8* bitcast ({ i8*, i8*, i32, i8* }* @_ZTIP7Subject to i8*), i8* null)
  unreachable

if.end:                                           ; preds = %if.then
  ret void
}
```

Source: Electronic Energy Meter Program:

Considering the intermediate code of Figure 4.18 where we can see the nodes identified by their labels and their predecessors we obtained the CFG presented in Figure 4.19.

Figure 4.19 - “update” correspondent CFG



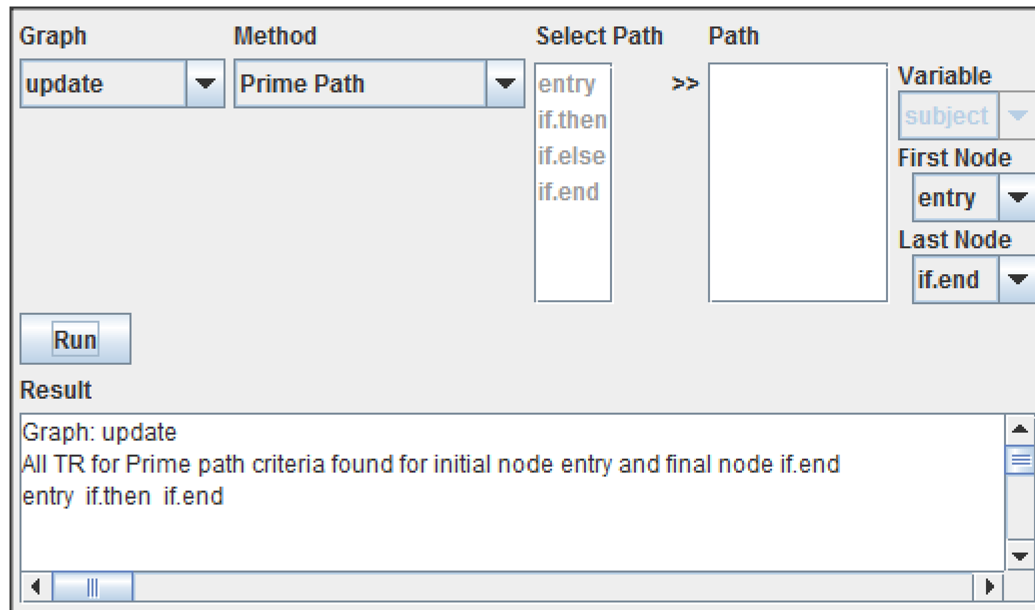
Source: The Author

Using the data described on this section, Figures 4.20 to 4.27 are the results obtained by the prototype for all the criteria available on the tool.

Figure 4.20 - Execution of Simple Path method for ”update” graph considering nodes “entry” and “if.end”

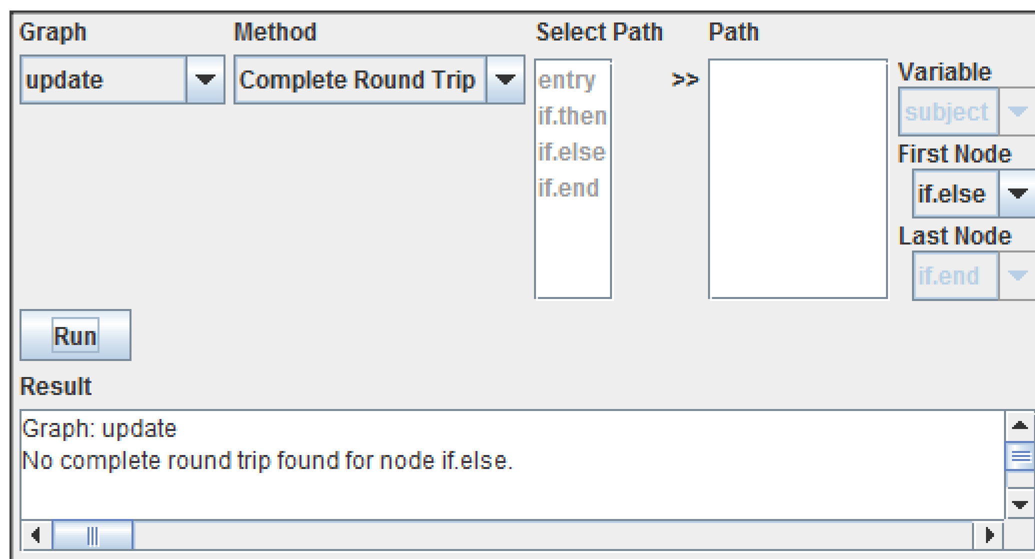
Source: The Author

Figure 4.21 - Execution of Prime Path method for “update“ graph, considering the nodes “entry” and “if.end”



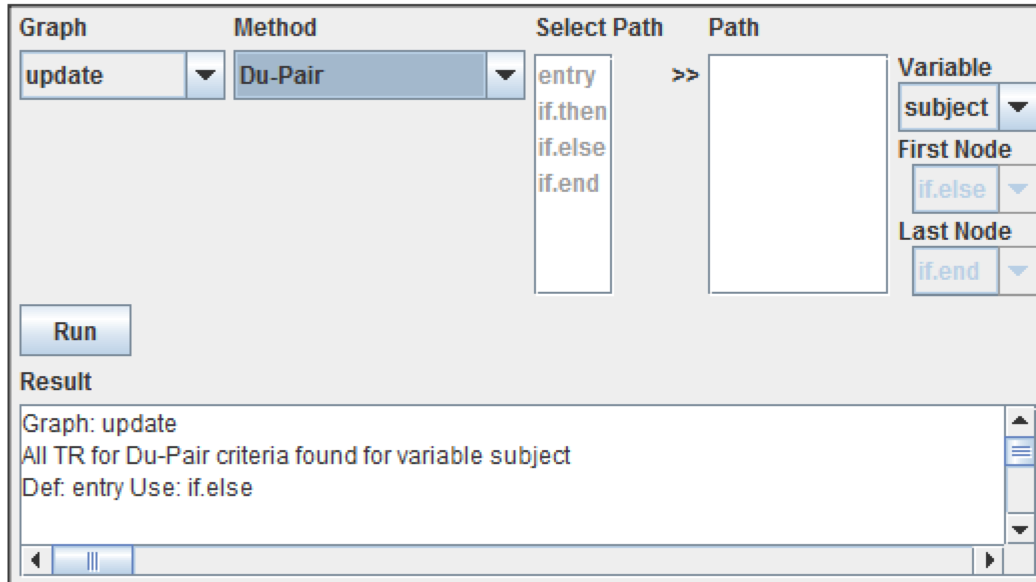
Source: The Author

Figure 4.22 - Execution of Complete Round Trip method for “update“ graph, considering the node “if.else”



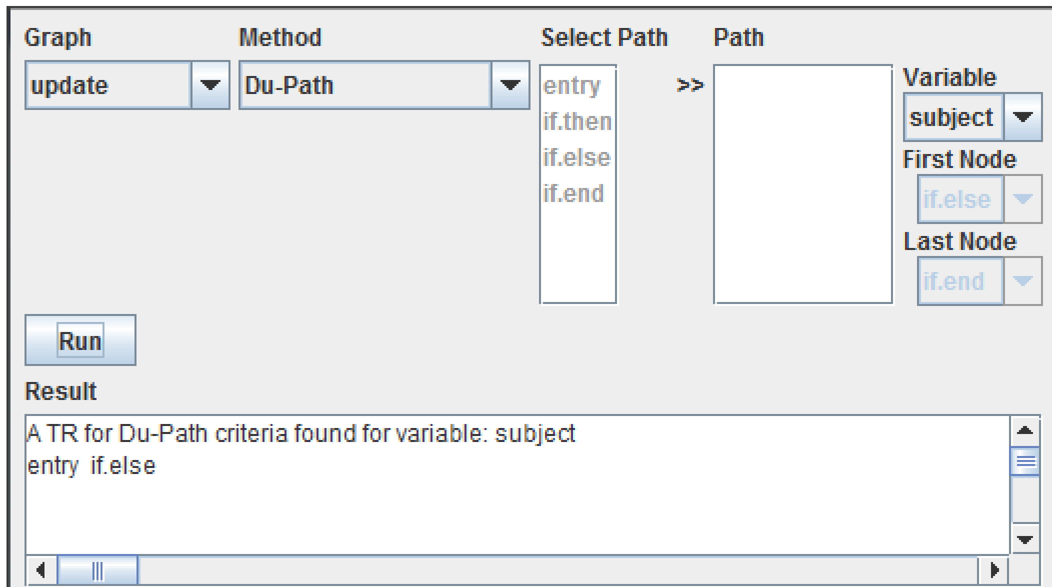
Source: The Author

Figure 4.23 - Execution of Du-Pair method for “update“ graph, considering the variable “subject”



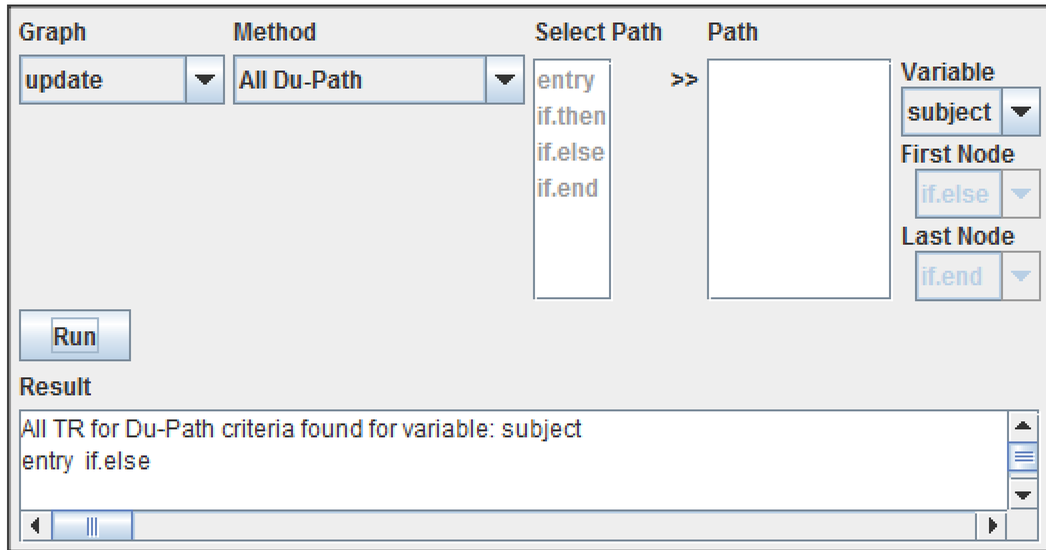
Source: The Author

Figure 4.24 - Execution of Du-Path method for “update“ graph, considering the variable “subject”



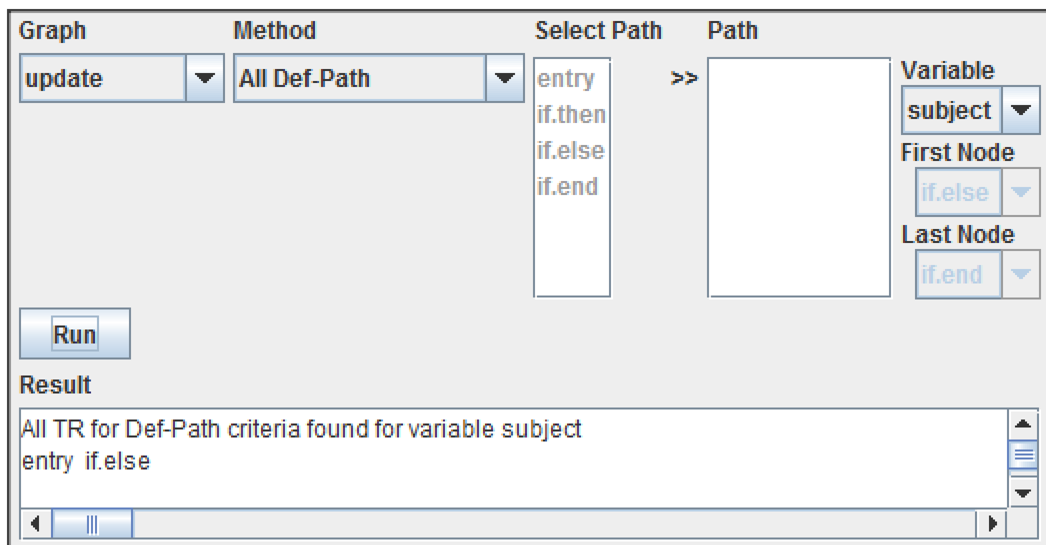
Source: The Author

Figure 4.25 - Execution of All Du-Path method for “update“ graph, considering the variable “subject”



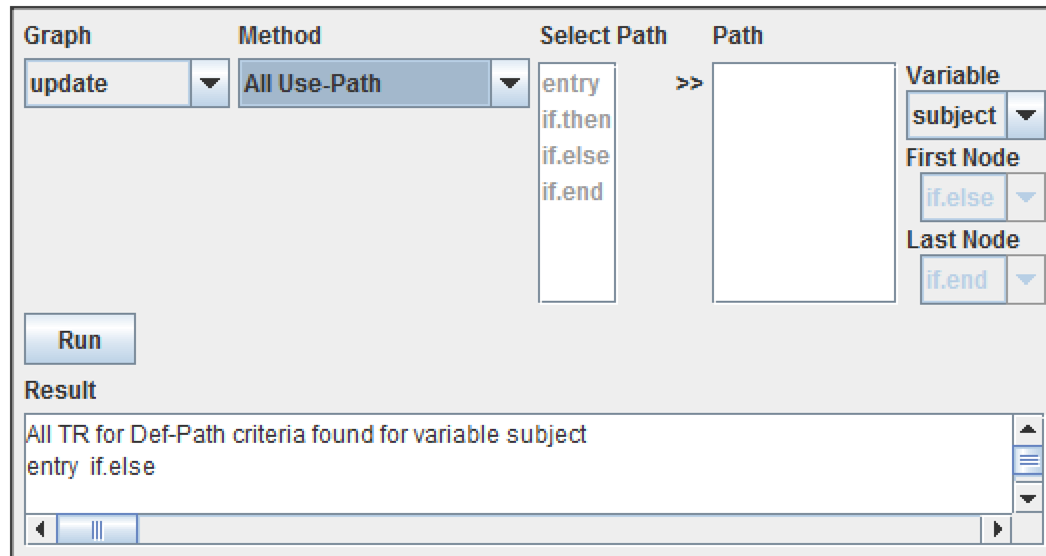
Source: The Author

Figure 4.26 - Execution of All Def-Path method for “update“ graph, considering the variable “subject”



Source: The Author

Figure 4.27 - Execution of All Use-Path method for “update“ graph, considering the variable “subject”



Source: The Author

4.4.1.3 Third Example

The third example is an execution of the prototype tool using as input file the intermediate code generated for class AT25DF321A of the embedded software. The method chosen to be validated on the next steps was the “_eraseMemory” for being easily representable.

Figure 4.28 - “eraseMemory” Source Code

```
void AT25DF321A::_eraseMemory() {
    uint32_t i = 0;
    uint32_t memorySize = configureSystem().memorySize;
    for (i=0; i < memorySize; i++) {

        _eraseMemData(i);
    }
    ;
}
```

Source: Electronic Energy Meter Program

The correspondent intermediate code generated by LLVM from “_eraseMemory” method can be visualized on Figure 4.29 (part of the code was omitted since it was not essential for this purpose).

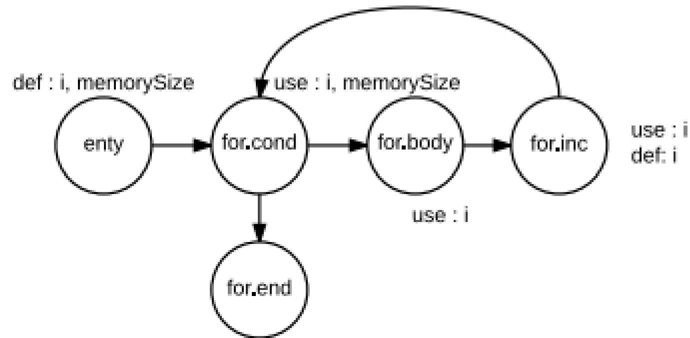
Figure 4.29 - “eraseMemory” Intermediate Code

```
define void @_ZN10AT25DF321A12_eraseMemoryEv(%class.AT25DF321A* %this) uwtable align 2 {
entry:
  %this.addr = alloca %class.AT25DF321A*, align 8
  %i = alloca i32, align 4
  %memorySize = alloca i32, align 4
  %coerce = alloca %struct.SystemConfigurationStruct, align 4
  %tmp = alloca { i64, i32 }
  store %class.AT25DF321A* %this, %class.AT25DF321A** %this.addr, align 8
  %this1 = load %class.AT25DF321A** %this.addr
  store i32 0, i32* %i, align 4
  %call = call { i64, i32 } @_Z15configureSystemv()
  store { i64, i32 } %call, { i64, i32 }* %tmp
  %0 = bitcast { i64, i32 }* %tmp to %struct.SystemConfigurationStruct*
  %1 = load %struct.SystemConfigurationStruct* %0, align 1
  store %struct.SystemConfigurationStruct %1, %struct.SystemConfigurationStruct* %coerce
  %memorySize2 = getelementptr inbounds %struct.SystemConfigurationStruct* %coerce, i32 0, i32 1
  %2 = load i32* %memorySize2, align 4
  store i32 %2, i32* %memorySize, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %3 = load i32* %i, align 4
  %4 = load i32* %memorySize, align 4
  %cmp = icmp ult i32 %3, %4
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %5 = load i32* %i, align 4
  call void @_ZN10AT25DF321A13_eraseMemDataEj(%class.AT25DF321A* %this1, i32 %5)
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %6 = load i32* %i, align 4
  %inc = add i32 %6, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret void
}
```

Source: Electronic Energy Meter Program

Considering the intermediate code of Figure 4.28 where we can see the nodes identified by theirs labels and their predecessors we obtained the Control Flow Graph (CFG) of Figure 4.29 for the method “_eraseMemory” of C++ class AT25DF321A.

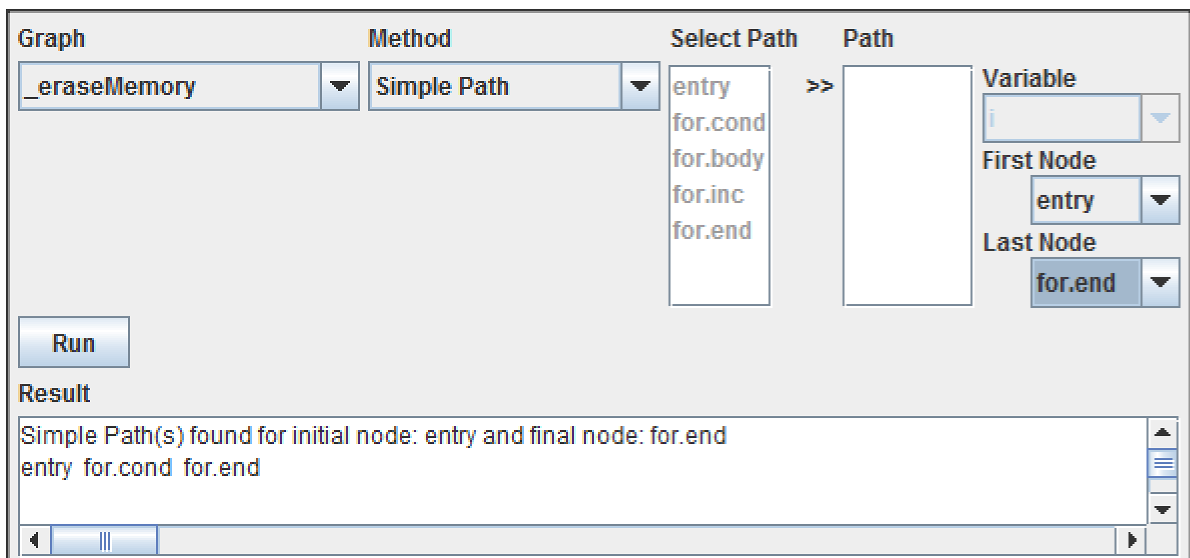
Figure 4.30 - “_eraseMemory” correspondent CFG



Source: The Author

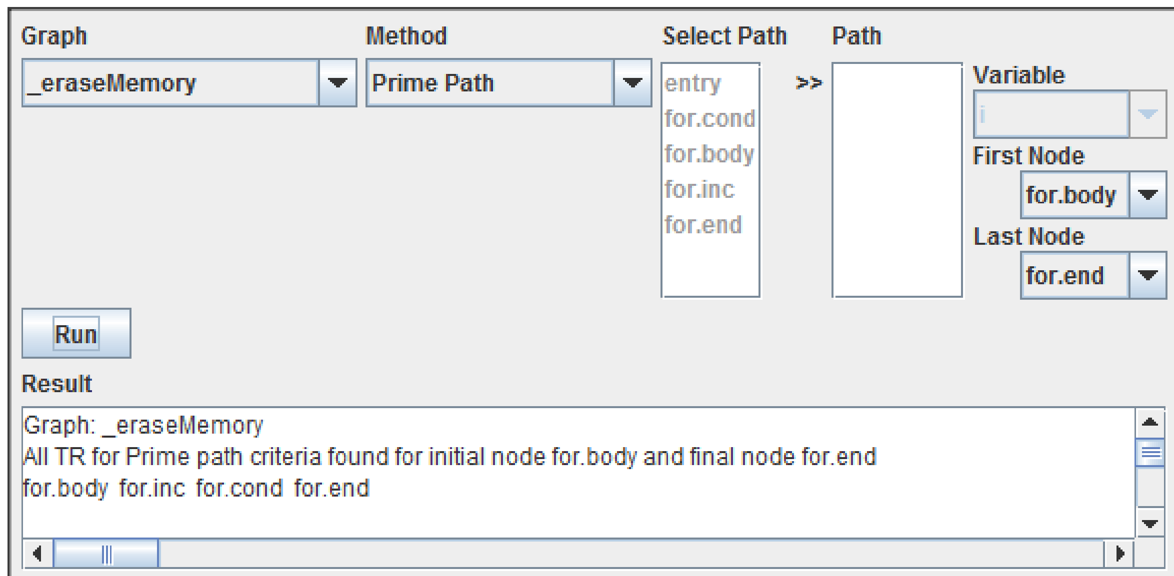
Using the data described on this section the Figures 4.30 to 4.38 are the results obtained by the prototype for all the criteria available on the tool.

Figure 4.31 - Execution of Simple Path method for “_eraseMemory“ graph, considering the nodes “entry” and “for.end”



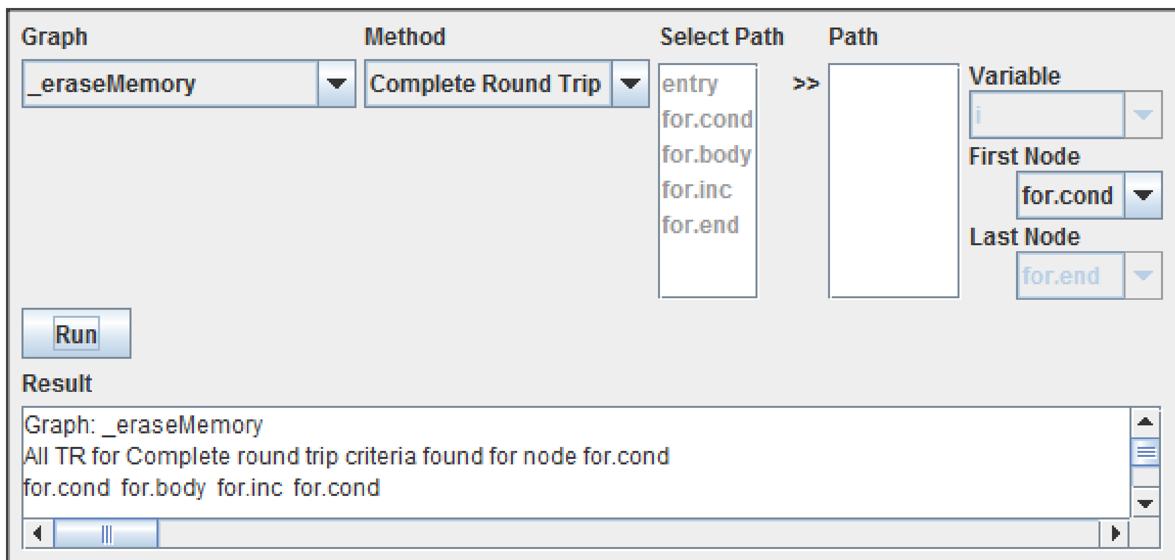
Source: The Author

Figure 4.32 - Execution of Prime Path method for “_eraseMemory“ graph, considering the nodes “for.body” and “for.end”



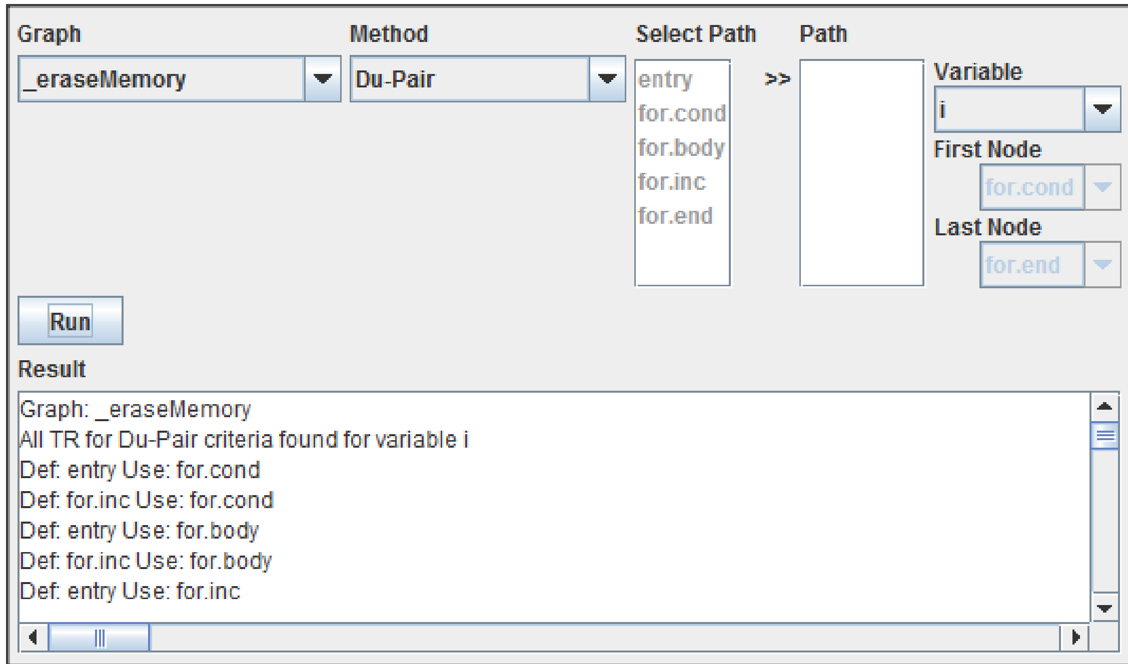
Source: The Author

Figure 4.33 - Execution of Complete Round Trip method for “_eraseMemory“ graph, considering the node “for.cond”



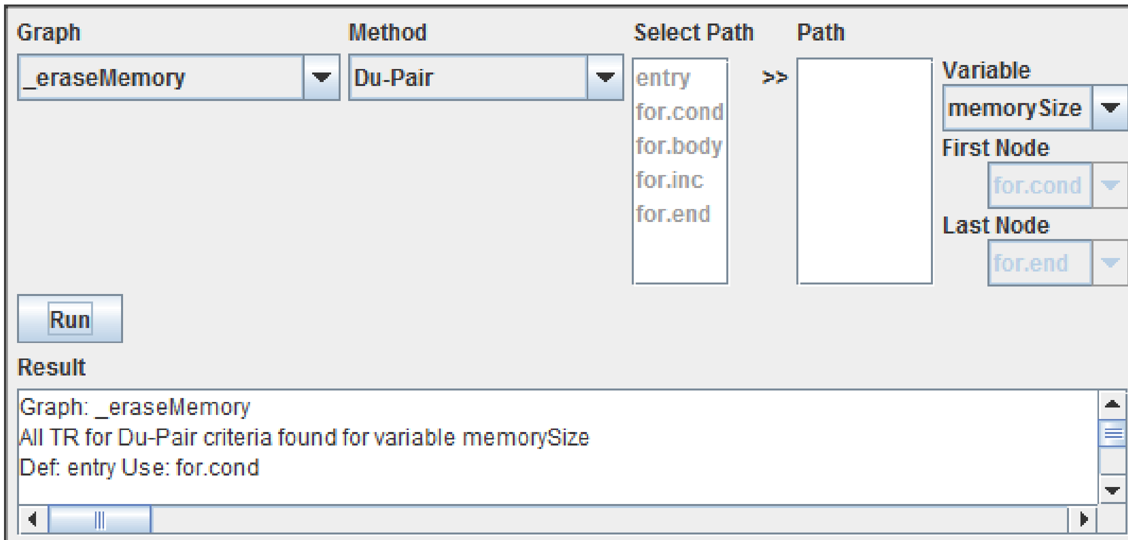
Source: The Author

Figure 4.34 - Execution of Du-Pair method for "_eraseMemory" graph, considering the variable "i"



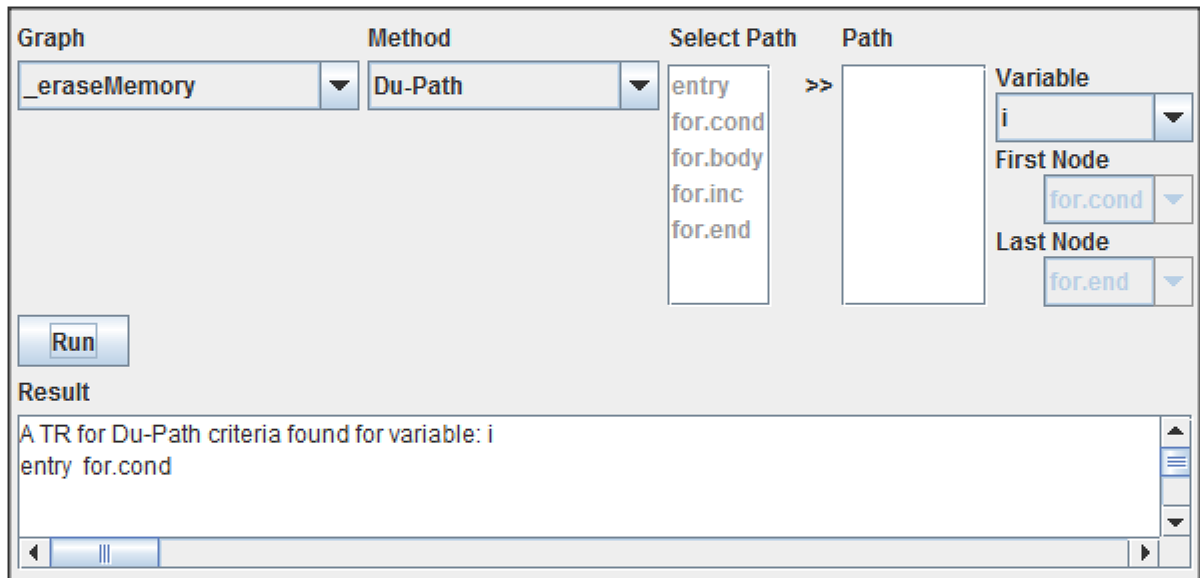
Source: The Author

Figure 4.35 - Execution of Du-Pair method for "update" graph, considering the variable "memorySize"



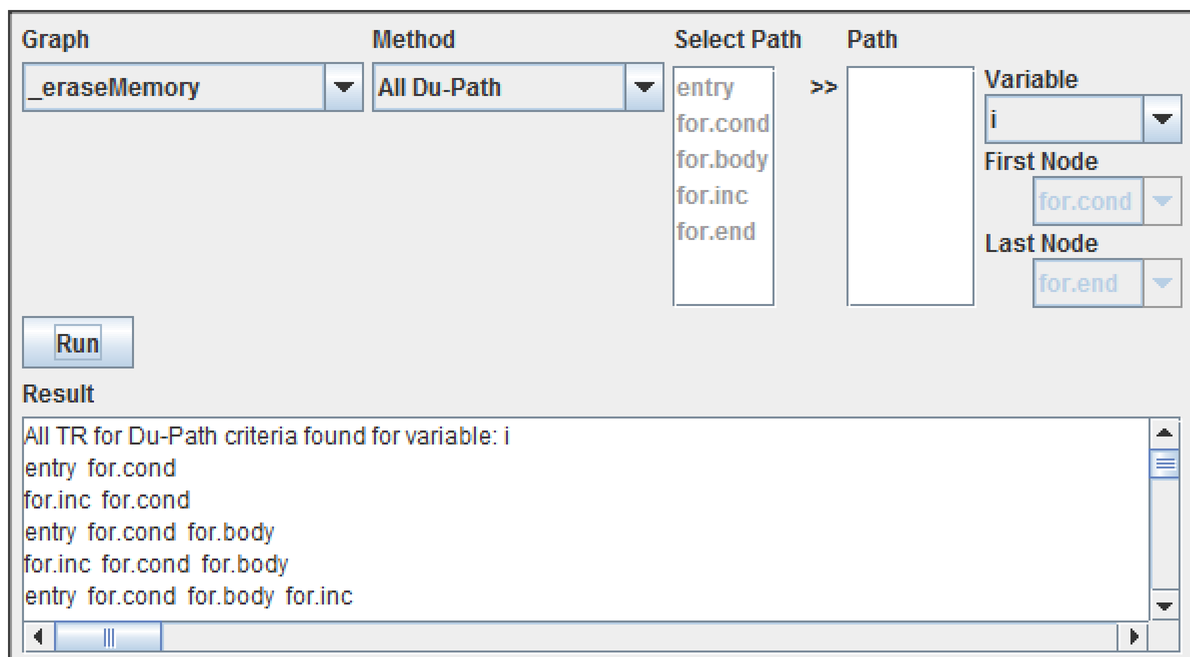
Source: The Author

Figure 4.36 - Execution of Du-Path method for “_eraseMemory“ graph, considering the variable “i”



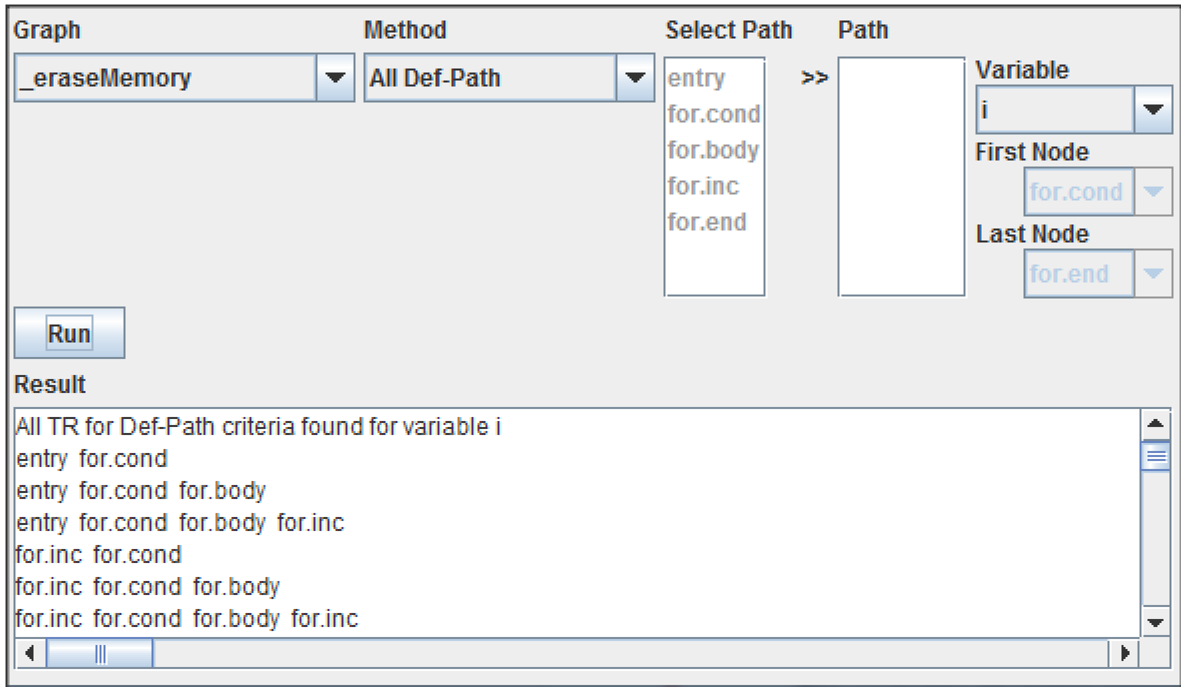
Source: The Author

Figure 4.37 - Execution of All Du-Path method for “_eraseMemory“ graph, considering the variable “i”



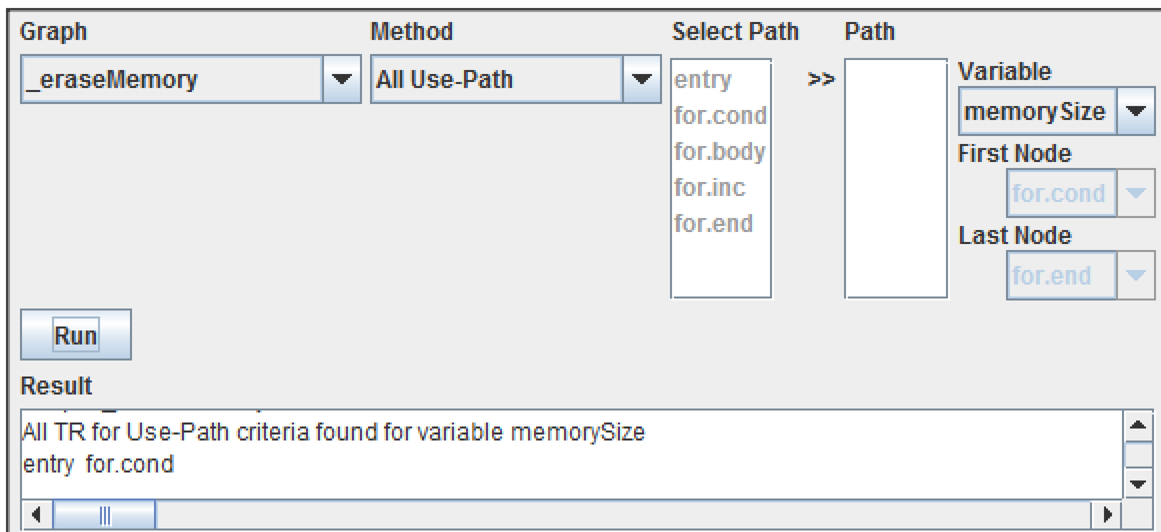
Source: The Author

Figure 4.38 - Execution of All Def-Path method for “_eraseMemory“ graph, considering the variable “i”



Source: The Author

Figure 4.39 - Execution of All Use-Path method for “_eraseMemory“ graph, considering the variable “memorySize”



Source: The Author

4.4.1.4 Fourth Example

The forth example is an execution of the prototype tool using as input file the intermediate code generated for class SPI of the embedded software. The method chosen to be validated on the next steps was the “getByte” for being easily representable.

Figure 4.40 - “getByte” Source Code

```
Tbyte SPI::getByte(vector<Tbit> bits) {
    Tbyte byte = 0;
    for (Tbyte i = 0; i < BYTE_SIZE; i++) {
        // Since variable byte is initialized with '0',
        // just set a bit if it is '1'
        if (bits[i]) {
            setBit(byte, 1, BYTE_SIZE - i - 1);
        }
    }
    return(byte);
}
```

Source: Electronic Energy Meter Program

The correspondent intermediate code generated by LLVM from “getByte” method can be visualized on Figure 4.40 (part of the code was omitted since it was not essential for this purpose).

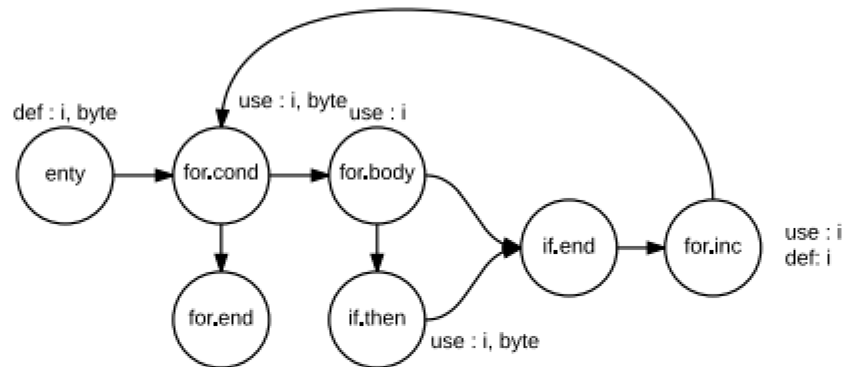
Figure 4.41 - “getByte” Intermediate Code

```
define zeroext i8 @_ZN3SPI7getByteESt6vectorIbSaIbEE(%class.SPI* %this,
                            %"class.std::vector"* %bits) uwtable align 2 {
entry:
  %this.addr = alloca %class.SPI*, align 8
  %byte = alloca i8, align 1
  %i = alloca i8, align 1
  %coerce = alloca %"struct.std::_Bit_reference", align 8
  store %class.SPI* %this, %class.SPI** %this.addr, align 8
  %this1 = load %class.SPI** %this.addr
  store i8 0, i8* %byte, align 1
  store i8 0, i8* %i, align 1
  br label %for.cond
for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i8* %i, align 1
  %conv = zext i8 %0 to i32
  %cmp = icmp slt i32 %conv, 8
  br i1 %cmp, label %for.body, label %for.end
for.body:                                         ; preds = %for.cond
  %1 = load i8* %i, align 1
  %conv2 = zext i8 %1 to i64
  %call = call { i64*, i64 } @_ZNSt6vectorIbSaIbEEixEm(%"class.std::vector"* %bits, i64 %conv2)
  %2 = bitcast %"struct.std::_Bit_reference"* %coerce to { i64*, i64 }*
  %3 = getelementptr { i64*, i64 }* %2, i32 0, i32 0
  %4 = extractvalue { i64*, i64 } %call, 0
  store i64* %4, i64** %3, align 1
  %5 = getelementptr { i64*, i64 }* %2, i32 0, i32 1
  %6 = extractvalue { i64*, i64 } %call, 1
  store i64 %6, i64* %5, align 1
  %call3 = call zeroext i1 @_ZNKSt14_Bit_referencecvbEv(%"struct.std::_Bit_reference"* %coerce)
  br i1 %call3, label %if.then, label %if.end
if.then:                                          ; preds = %for.body
  %7 = load i8* %i, align 1
  %conv4 = zext i8 %7 to i32
  %sub = sub nsw i32 8, %conv4
  %sub5 = sub nsw i32 %sub, 1
  %conv6 = trunc i32 %sub5 to i8
  call void @_Z6setBitRhbh(i8* %byte, i1 zeroext true, i8 zeroext %conv6)
  br label %if.end
if.end:                                           ; preds = %if.then, %for.body
  br label %for.inc
for.inc:                                          ; preds = %if.end
  %8 = load i8* %i, align 1
  %inc = add i8 %8, 1
  store i8 %inc, i8* %i, align 1
  br label %for.cond
for.end:                                          ; preds = %for.cond
  %9 = load i8* %byte, align 1
  ret i8 %9
}
```

Source: Electronic Energy Meter Program

Considering the intermediate code of Figure 4.40 where we can see the nodes identified by theirs labels and their predecessors we obtained the Control Flow Graph (CFG) of Figure 4.41 for the method “getByte” of C++ class SPI.

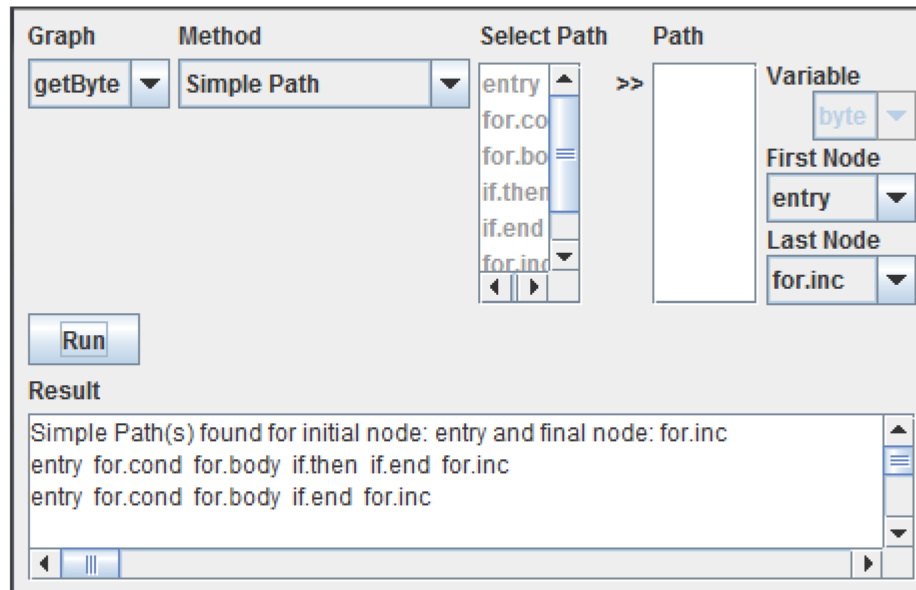
Figure 4.42 - “getByte” correspondent CFG



Source: The Author

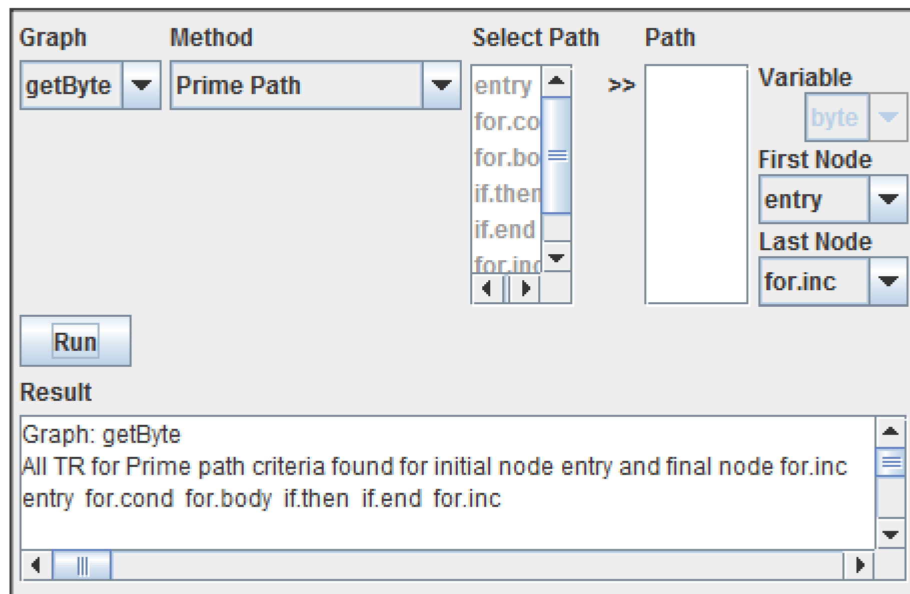
Using the data described on this section the Figures 4.42 to xxx are the results obtained by the prototype for all the criteria available on the tool.

Figure 4.43 - Execution of Simple Path method for “getByte “ graph, considering the nodes “entry” and “for.inc”



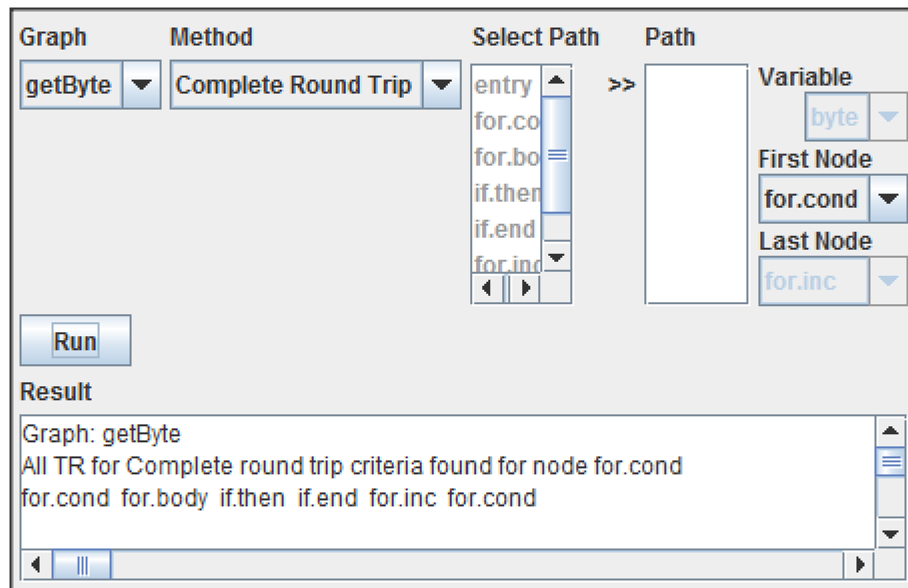
Source: The Author

Figure 4.44 - Execution of Prime Path method for “getByte “ graph, considering the nodes “entry” and “for.inc”



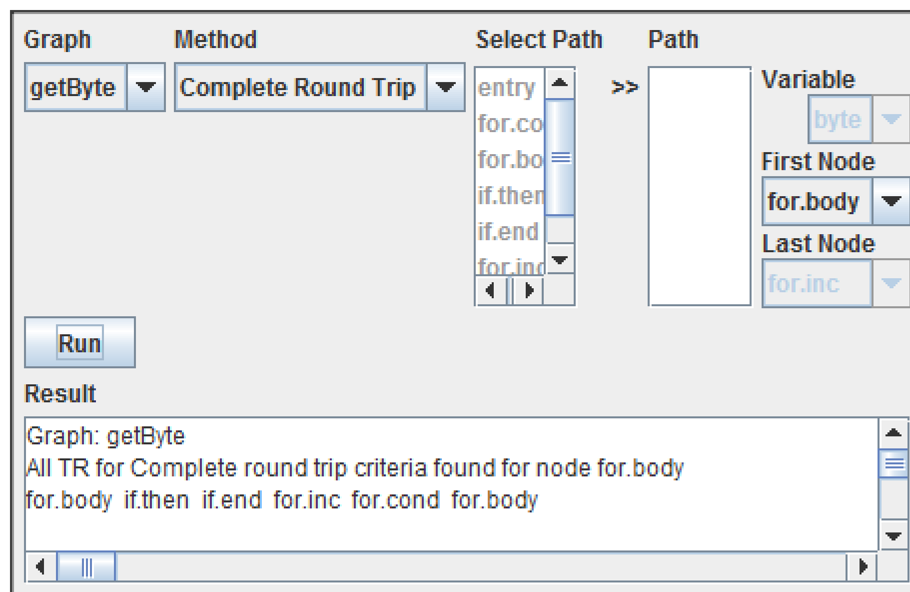
Source: The Author

Figure 4.45 - Execution of Complete Round Trip method for “getByte “ graph, considering the node “for.cond”



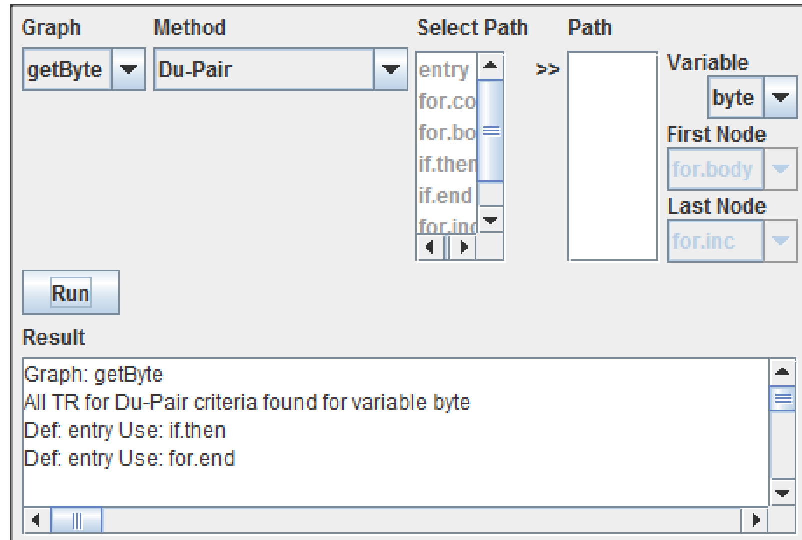
Source: The Author

Figure 4.46 - Execution of Complete Round Trip method for “getByte “ graph, considering the node “for.body”



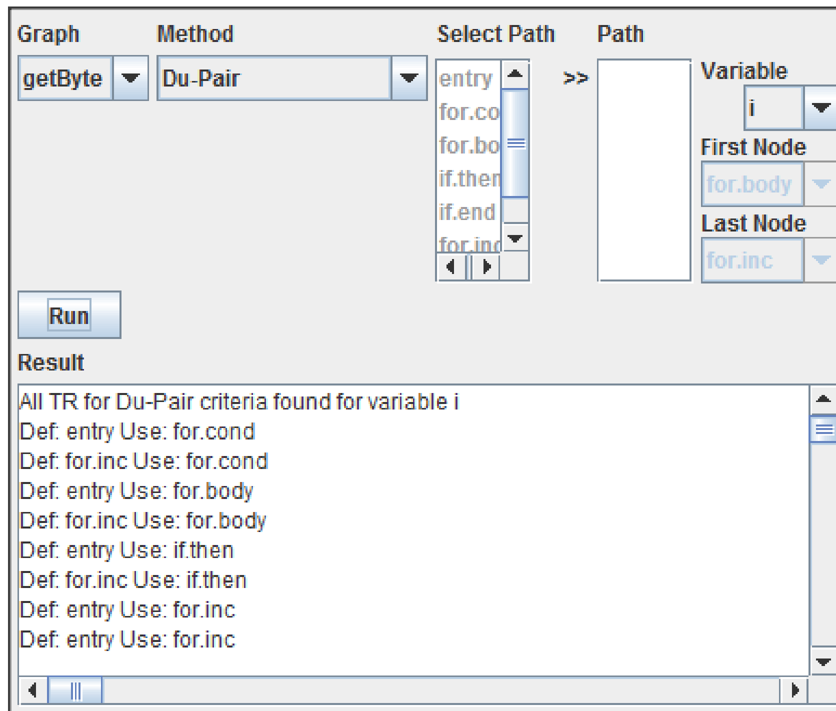
Source: The Author

Figure 4.47 - Execution of Du-Pair method for “getByte “ graph, considering the variable “byte”



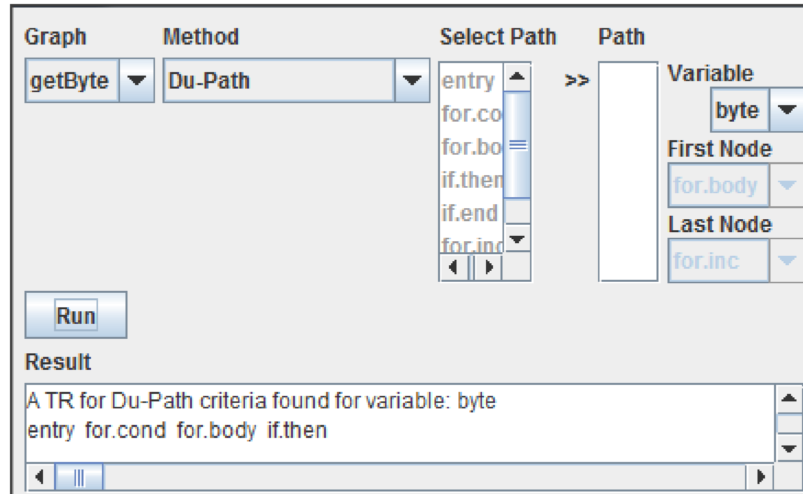
Source: The Author

Figure 4.48 - Execution of Du-Pair method for “getByte “ graph, considering the variable “byte”



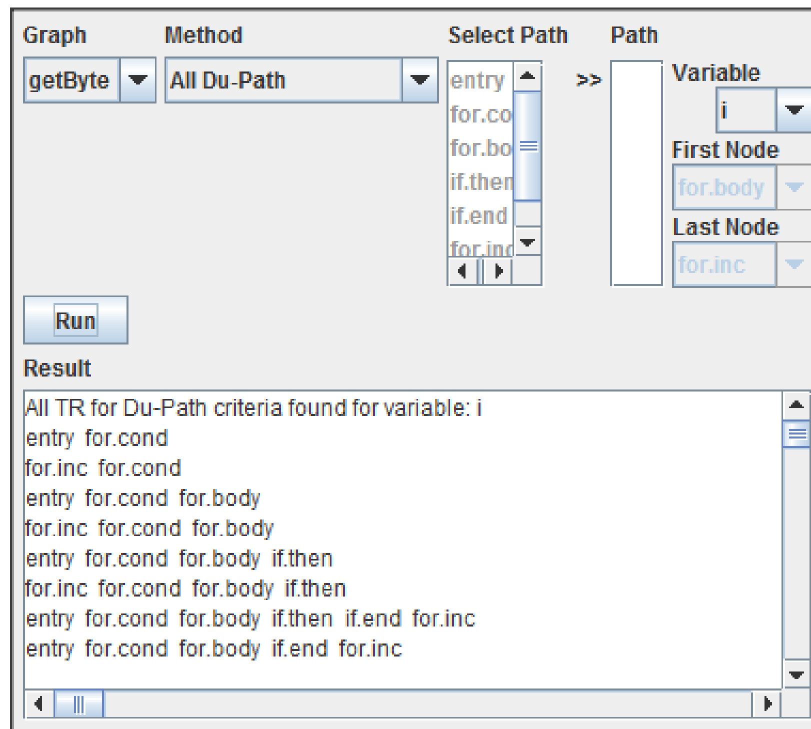
Source: The Author

Figure 4.49 - Execution of Du-Pair method for “getByte “ graph, considering the variable “byte”



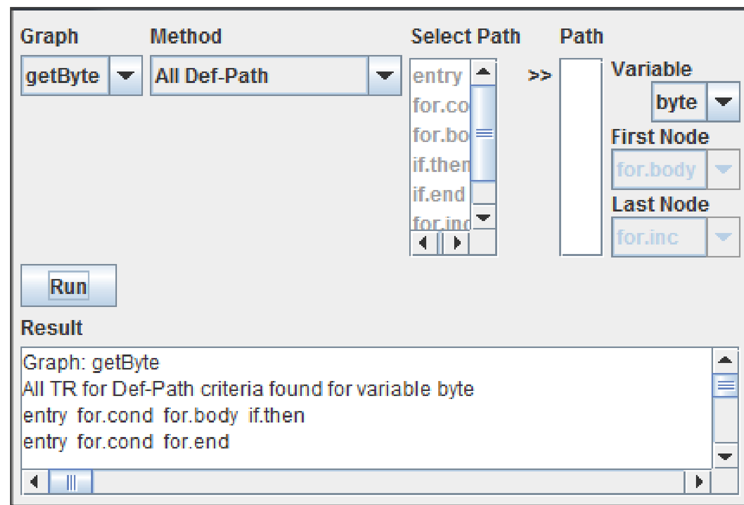
Source: The Author

Figure 4.50 - Execution of All Du-Path method for “getByte “ graph, considering the variable “byte”



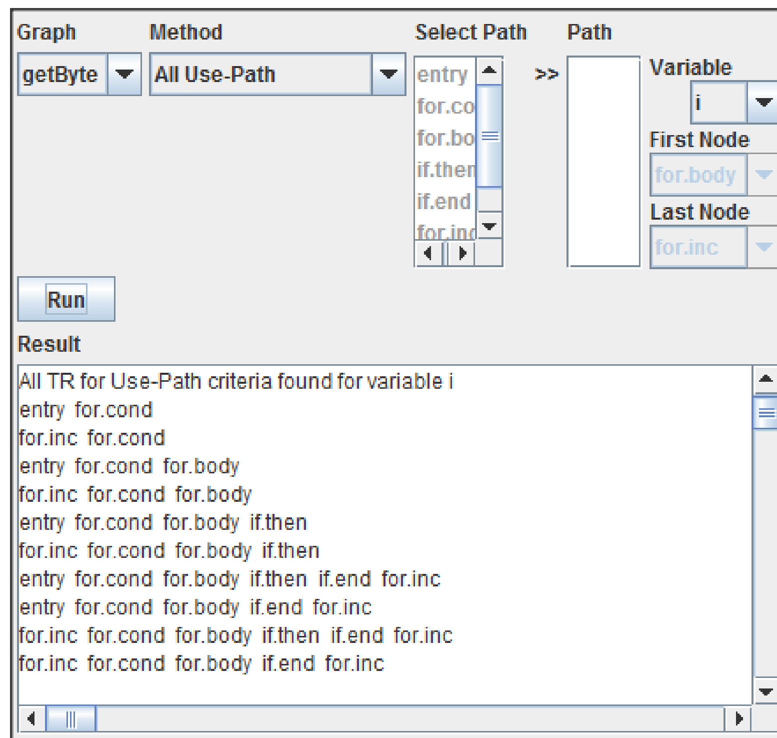
Source: The Author

Figure 4.51 - Execution of All Def-Path method for “getByte “ graph, considering the variable “byte”



Source: The Author

Figure 4.52 - Execution of All Def-Path method for “getByte “ graph, considering the variable “byte”



Source: The Author

5 FINAL REMARKS

This work was developed in order to exercise the importance of testing criteria. Considering the huge need of testing software, since they are present everywhere in modern life, methods to increase the performance of testing are extremely necessary to make testing feasible.

The proposal was to create an application able to apply some of the existent code coverage criteria over a generic object oriented program previously compiled by LLVM and return Test Requirements found for each criteria. Basically we should have a program which could read the intermediate file generated by LLVM (which read C++ programs) and allow the user to run Intramethod tests over the CFG of each method found in the input file.

Considering the huge efforts needed to build an application able to read LLVM intermediate file and extract a Control Flow Graph from this intermediate code, the prototype has limited features. The current program is able to identify and return, for each method of C++ input files, the Test Requirements for a certain set of criteria selected. Applying coverage criteria over the Control Flow Graphs and considering the Test Requirements found is still an opportunity of improvement for this tool.

5.1 Proposed Features

The tool developed on this work is able to generate the Test Requirements needed for coverage criteria and can be complemented with the development of Test Requirements for additional criteria, such as Node Coverage or Edge Coverage Criteria. It is also an important improvement to develop the methods that perform coverage analysis for each criterion, that is, evaluate the coverage of the set of test requirements by a given set of tests, and provide the user a visual understanding of the generated CFGs, adding this information to the user interface.

Furthermore, using this set of basic methods, actual object-oriented coverage analysis can also be implemented, providing the tester/developer more information about the quality of its test set.

REFERENCES

AMMAN, Paul; OFFUT, Jeff. Introduction to Software Testing. New York: Cambridge University Press, 2008.

Clang Compiler User's Manual. Available at: <clang.llvm.org/docs/UsersManual.html>. Accessed in 26 jun. 2015.

The LLVM Compiler Infrastructure. Available at <llvm.org>, Accessed in 21 jun. 2015.

GOMES, Humberto V..Metodologia de Projeto de Software Embarcado Voltada ao Teste. Thesis presented as partial pre requirement for Masters degree – Federal University of Rio Grande do Sul, Informatics Institute, Porto Alegre, may. 2010. p. 47