Generating Control Flow Graphs from NATURAL

Strauss Cunha Carvalho * †, Renê Esteves Maria *, Leonardo Schmitt †, and Luiz Alberto Vieira Dias *
*Computer Science Division
Aeronautics Institute of Technology - ITA, São José dos Campos, BRA
Email: (rene, vdias) @ita.br
†Brazilian Federal Service of Data Processing - SERPRO, Rio de Janeiro, BRA
Email: (strauss.carvalho, leonardo.schmitt) @serpro.gov.br

***Abstract*—This work aims to generate White-box Test Cases from a mainframe NATURAL code fragment, using the Control Flow Graphs Technique. Basically, it enables a code fragment analysis and generates its control flow represented by a Graph perspective. As a consequence, this work provides the automation of White-box Test Cases. Furthermore, this work contributes to decrease the degree of difficult to execute White-box Tests, within a Mainframe environment. Also, it brings a significant contribution related to the execution time, inherent to the software testing. At the end, this work adds testing expertise to NATURAL development teams. Most of times, there is no enough available time to test all possible paths in a algorithm, even it is a simple application. Based on it, this work provides a speed-up test cases generation, that empowered the team decision regarding which test cases are more relevant to be performed.**

Keywords*—*Software Testing and Engineering, Software Quality, Control Flow Technique.

1. Introduction

According to Molinari [1], there is a relationship between the developed systems quality and their respective testing activities. Therefore, the software testing process becomes essential for product quality assurance or developed service. Consequently, it is possible to observe in public or private organizations, an increasing demand for the software development among the most diverse segments such as commercial, industrial, government, military, and scientific.

Usually, testing software activities increase the development cost. According to Pressman [2], is possible to see a direct relation between time to defect detection with the development cost. In other words, when more early a defect detection occurs, the saving is up to hundred times lesser, if comparing to the final development phase.

Therefore, there is no enough time to test all possible paths in an algorithm, even for simple applications. Due financial limitations, the spent time testing activities can be directly add to the development cost. It increases the final product price [2].

Through the years, the software testing area has been developed models, methods, and techniques, that match the ideal cost/benefit [7]. According to Pressman [2], it is a way to identify internal software defects, named White-box Tests (WbT). Within the WbT is possible to use the Control Flow Technique (CFT), it enables a clear code review.

Based on this context, this work has developed a WbT tool, using a variation of the CFT, named Control Flow Graphs (CFG). The developed tool, which works within Mainframe NATURAL code fragments [3], aims the creation test cases automation.

Moreover, it generates a graph that represents the code flow. The created graph intends to decrease the code analysis difficulty. At the end, this work tries to improve the software quality, reducing time to create test case, optimizing time to perform tests, and bringing the defect detection to early development stages.

This work is organized in the following structure: Section two is devoted to describe the work background regarding the testing software area; Section three describes the WbT proposed tool. Section four discusses and presents obtained results from two experiments where the proposed tool was applied. Finally, in section five concluding remarks are presented.

2. Software Testing

This section describes relevant concepts that involve this work. Conforming the standard IEEE 610.12-1990 [6], the test activity is defined by the system or component operation process, under specific conditions. Indeed, it requires the registration and observation regarding collected results, enabling the system or component evaluation.

According to Crispin and Gregory [7], the agile software development requires a new concept to the traditional tester role, named agile tester. The agile tester is a team member that comprises an agile development team (also know as cross-functional team). Also, the agile tester has to be involved with the business and technical areas, in order to improve and refine the product requirements. It is fundamental to boost the development in the right direction.

Withal, the main agile tester skill is turning product requirements into automated tests. In most cases, they are experienced exploratory testers trying to understand user wishes and needs. Therefore, the software quality is a common goal to the development team [7].

A. Agile Testing Quadrants

The agile testing quadrants has been used as a guide to the development team. Initially, the quadrants were created by Brian Marick, presenting a numerated matrix divided into four cells. Each cell number does not imply a required order to be followed [7].

According to Crispin e Gregory [7], the agile testing quadrant is a simple taxonomy that enables a test planning to the agile development team. Therefore, it provides a minimum

required self-organization to start the development within each sprint. The Figure 1 is showing the agile testing quadrants. This work is based on the Q1 and Q2.

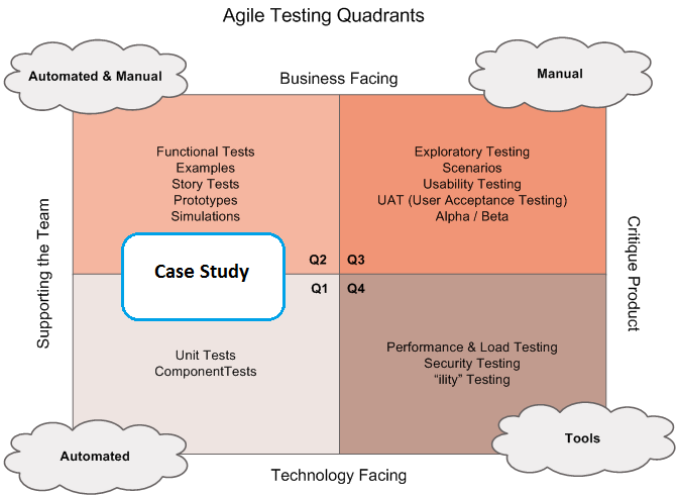

Fig. 1. The Agile Testing Quadrants - adapted from [7]

The Q1 focuses the prime testing techniques used in the agile development, like the Test-driven Development (TDD) [7]. The TDD provides to team members the possibility to develop features without worrying about later changes. Consequently, it improves the development and application quality. Moreover, it helps the team makes better decisions regarding its project design.

As reported by Meszaros [8], this quadrant has two practices: 1) **unit test**, to validate a smaller part of the application, that means objects or methods; 2) **component test**, to validate the major part of the application as a set of classes that provides the same task. Usually, the Q1 tests are developed with xUnit tools, they aim to measure the internal software quality.

Unit tests are not design by costumers, it will not understand the internal development aspects. They must be performed within a continue integration approach. In fact, it provides the source code quality assurance [9]. Therefore, when the team members are writing their unit tests, their testing functionality occurs before its own existing.

This work is concentrated in the Q1 and Q2. After the algorithm writing, this work contributes to its graphical visualization of possible paths, and to automatic generating its respective test cases. Thus, functional testings are performed focusing on the customer understanding that establishes quality criteria.

At the end, this work provides a friendly language that could be used by all involved resources. In other words, tests are written in such manner that business experts can understand the implemented features through this friendly language.

B. White-box Testing

According to Myers [10], the tests based on implementation are named White-box Testing (WbT) and Structural Testing. The test selection is based on information obtained from source code, in order to perform different software parts disregarding its specification.

The WbT is based on the internal software architecture. It engages techniques to identify internal structure defects. According to Bartié [11], to perform WbTs is required agile testers with enough knowledge about the technology used. Also, the internal software architecture.

Conform Pressman [2], to execute the WbT some techniques are used to identify a software internal structure defects, like control flow and data stream.

C. Control Flow Technique

The Control Flow Technique (CFT) shows the possible paths in an algorithm by symbolic representations. Usually, graphs are used to visualize the software logic controls. Applying the graph usage, this technique can be known as Control Flow Graphs (CFG) [2].

The Figure 2 is showing a CFG sample. Each vertex represents algorithm logic controls, each arc corresponds the possible paths between logic controls. There are five vertexes and seven arcs in the Figure 2.

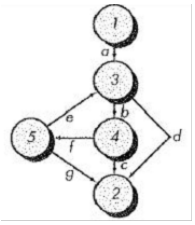

Fig. 2. A Control Flow Graph from [2]

According to Pressman [2], the CFG creation can be a hard activate, when the analysed algorithm has compound conditions. That means, when exist boolean operations as or, and, not-and, and not-or. Therefore, this work is using the CFG and tries to reduce its creation complexity.

D. Test Cases

Conforming to Craig and Jaskiel [12], test cases are comprised by inputs, outputs, execution restriction rules, and expected results/behaviors. It enables document and test applications, under predefined conditions.

As reported by Heineberg [13], the test case main goal is formally communicate and identify software defects. It allows the software quality evaluation.

However, there is a significant effort for manual test case writing. Especially when it is trying to cover all possible paths. It consumes more development time, consequently it increases the final project cost [7].

Thereby, some approaches were proposed tools for generating test cases, from certain specifications. This work has been investigated the proposed approaches from Santiago [??] and Arantes [??] apud Marinke [??].

Based on those approaches, this work propounds a WbT tool based on CFG. The next section describes the proposed tool. Basically, it uses a NATURAL fragment code for generating its flow graph and test cases.

3. Proposed Tool

Initially, this work was looking to source codes written in NATURAL Language for Mainframe environments [3]. It was observing difficulties faced by organizations where trying to perform WbT in this type of source code. Moreover, there is a high cyclomatic complexity inherent in its legacy programs. In most cases, this type of language have no object-oriented design that provides this kind of situation.

The code fragments used in this work had been developed at the Brazilian Federal Service of Data Processing (SERPRO) [4]. It is relevant to highlight that keeping classified information, there are no confidential information and/or business aspects. This work is focusing in analysis of algorithms within their Mainframe context.

This proposed tool used two different code fragments as experiments. Both were exposed to the same path. Initially, their NATURAL source codes were read. Then, their flow graphs were automatic generated. Finally, their respective test cases were produced. This path can be divided into five steps, as the Figure 3 illustrates.

The Figure 3, shows a diagram containing the proposed tool five steps. Moreover, The Figure 3 clarifies the proposed tool execution path. Which starts from the source code fragment until the test cases output. Each one of the five steps are described within this section.

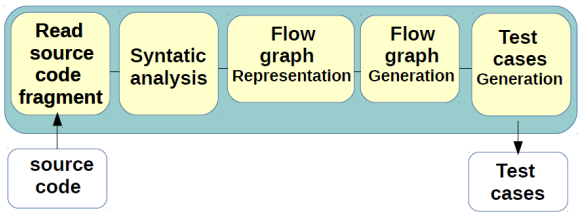

Fig. 3. The Proposed Tool Execution Path

The proposed tool was written in the Python programming [5]. Therefore, using Python native modules, the first step consists in read the input code line by line. It enables the next step the source code syntactic analysis.

The second step performs a syntactic analysis from the last step output. The syntactic analysis (also known as parsing) term refers the translation process between language programs. Based in a formal grammar, it aims to recognize the program syntactic structure.

After parsing and during the execution time, the third step converts the source code into a flow graph. Thus, the source code analyzed is represented through a flow graph. The adjacency matrix is used as a data structure for the graph representation. Also, this matrix has been stored in the main memory.

In the fourth stage, the previous graph representation is converted into a visual mode. Based on adjacency matrix stored in the main memory, this proposed tool is using Python graphs modules. It provides to the development team a quickly visualization of their implemented possible paths.

Using the third step provided matrix, the last step is devoted to generate the respective test cases corresponding to the input source code. Within this step, each graph arcs is visited, for each visit a test case is generated. It produces a complete path coverage to the analyzed code fragment.

At the end, the development team has to decide which test cases will be performed within the current sprint. In fact, to avoid an enormous amount of test cases, it is recommended to break down the functionality into smaller parts to be tested.

4. The Main Results

This section tackles the main results obtained from two conducted experiments. Moreover, it shows their NATURAL code fragments, as well their respective CFGs and Test Cases. There is a cyclomatic complexity crescent between both experiments. That means, the first experiment is more simple then the second experiment.

A. First Experiment

The Figure 4 is showing the source code fragment used in the first experiment. This fragment has two conditional structures in sequence, the IF and ELSE statements.

```
1  IF HIST-VIEW.NU-TERMINAL-IP  NE '                '
2    MOVE HIST-VIEW.NU-TERMINAL-IP              TO #NU-TERMINAL-IP(#I)
3  ELSE
4    MOVE HIST-VIEW.TERMINAL-USUARIO            TO #NU-TERMINAL-IP(#I)
5  END-IF
6  MOVE 'PEMP'                                  TO #IND-MUDANCA(#I)
7  MOVE HIST-VIEW.DA-EVENTOS(06)                TO #DATA-WORK
8  MOVE #DT-WORK-1                              TO #DATA-EVENTO(#I)
9  IF HIST-VIEW.IT-PORTE-EMPRESA= '01'
10   MOVE 'MICROEMPRESA'                        TO #ALTERACAO(#I)
11 ELSE
12   IF HIST-VIEW.IT-PORTE-EMPRESA= '03'
13     MOVE 'EMPRESA DE PEQUENO PORTE'          TO #ALTERACAO(#I)
14   ELSE
15     IF HIST-VIEW.IT-PORTE-EMPRESA= '05' OR= ' '
16       MOVE 'DEMAIS'                          TO #ALTERACAO(#I)
17     END-IF
18   END-IF
19 END-IF
```

Fig. 4. The Source Code Fragment used in the First Experiment

Following the proposed tool execution path, the Figure 5 illustrates the extracted CFG from the first code fragment (see Figure 4). Also, their test cases are showing in the Figure 6.

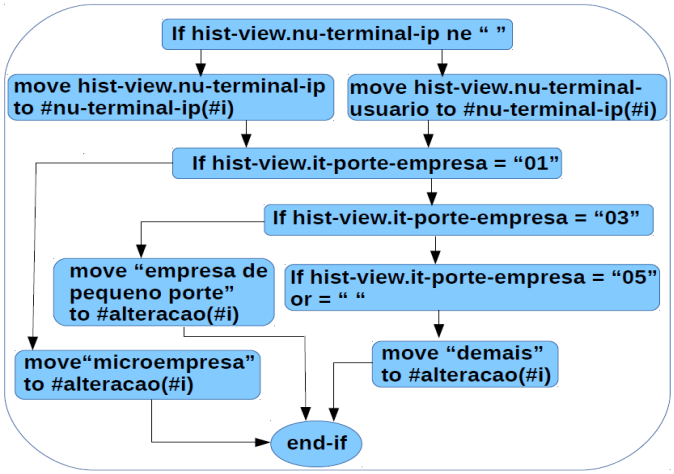

Fig. 5. The CFG from the First Experiment

	A	B	C	D	E	F
1		if hist-view.nu-terminal-ip ne " "	If hist-view.it-porte-empresa = "01"	If hist-view.it-porte-empresa = "03"	If hist-view.it-porte-empresa = "05"	Result
2	move hist-view.nu-terminal-ip to #nu-terminal-ip(#i)					
3	Hist-view.nu-terminal-usuario to #nu-termi-nal-ip(#i)					
4	move "Microempre-sa" to #alteracao(#i)					
5	move "empresa de pequeno porte" to #alteracao(#i)					
6	mode "demais" to #alteracao(#i)					

Fig. 6. The Test Cases from the First Experiment

B. Second Experiment

The second experiment was based in the source code fragment shown in the Figure 7. It is an algorithm with more cyclomatic complexity then the first experiment.

Looking to its line nineteen (see Figure 7), it is possible to observe multiple conditions using the logical operator AND. Furthermore, there are a dense sequence of IF and ELSE statements. It increases the amount of possible paths.

```
1  IF FCPJ_CO-EVENTOS-N(*) = 101 AND STATUS_CO-STATUS-PEDIDO EQ 83
2    IF  #AX-TEM-EV-INTERESSE-EST EQ 'S'
3      MOVE #CO-UF-AUX   TO #AX-CONV-PPA
4      PERFORM PESQUISAR-CNAE-INTERESSE-CONV
5      IF #AX-CNAE-INTERESSE-CONV-IC = 'S'
6        PERFORM ADICIONAR-ARRAY-CONV-PPA
7      END-IF
8    END-IF
9    IF  #AX-TEM-EV-INTERESSE-MUNIC EQ 'S'
10     MOVE #CO-MUN-AUX TO #AX-CONV-PPA
11     IF #AX-CONV-PPA  = #AX-ARRAY-CONVENENTES (*)
12       PERFORM ADICIONAR-ARRAY-CONV-PPA
13     END-IF
14   END-IF
15   IF #CO-UF-AUX EQ 'MG'
16     MOVE '31  ' TO #AX-CONV-PPA
17     PERFORM ADICIONAR-ARRAY-CONV-PPA
18   END-IF
19   IF  #CO-UF-AUX EQ 'MA' AND #AX-TEM-EV-INTERESSE-JUNTA = 'S'
20      MOVE '21  ' TO #AX-CONV-PPA
21      PERFORM ADICIONAR-ARRAY-CONV-PPA
22    END-IF
23    ESCAPE ROUTINE
24 ELSE
25    IF FCPJ_CO-EVENTOS-N(*) = 246
26      PERFORM SETAR-ARRAY-CONV-PPA-246
27    ELSE
28      PERFORM SETAR-ARRAY-CONV-PPA
29    END-IF
30 END-IF
31 PERFORM SETAR-ARRAY-CONV-INFO
```

Fig. 7. The Source Code Fragment used in the Second Experiment

Following the proposed tool execution path, the Figure 8 illustrates the extracted CFG from the second code fragment (see Figure 7). Also, their test cases are showing in the Figure 9. At the end, the second experiment had been generated eighty-eight test cases (see Figure 9).

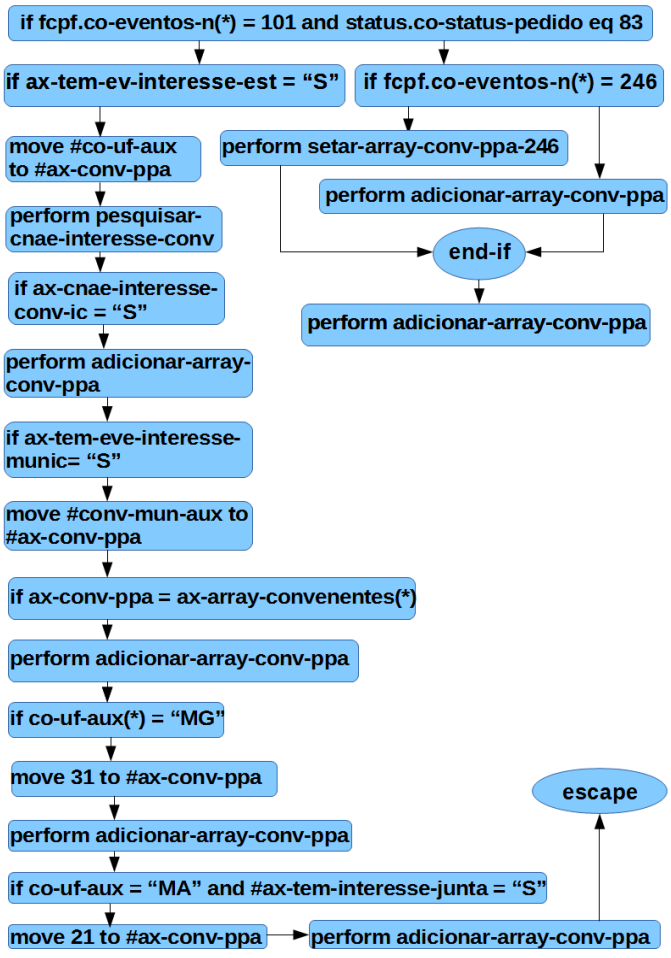

Fig. 8. The CFG from the Second Experiment

	A	B	C	D	E	F	G	H	I	J
1		if fcpj.co-eventos-n(*) = 101 and status.co-status-pedido eq 83	if ax-tem-ev-interesse-est = "S"	if #ax-cnae-interesse-conv-ic = "S"	if ax-tem-ev-interesse-munic = "S"	if #ax-conv-ppa = #ax-array-convenentes (*)	if #co-uf-aux = "MG"	if co-uf-aux eq "MA" and #ax-tem-interesse-junta = "S"	if fcpj.co-eventos-n(*) = 246	Result
2	move #co-uf-aux to #ax-conv-ppa									
3	perform pesqui-sar-cnae-inte-resse-conv									
4	perform adicio-nar-array-conv-ppa									
5	mun-aux to #ax-conv-ppa									
6	perform adicio-nar-array-conv-ppa									
7	move "31 " to #ax-conv-ppa									
8	perform adicio-nar-array-conv-ppa									
9	move "21 " to #ax-conv-ppa'									
10	perform adicio-nar-array-conv-ppa									
11	perform setar-ar-ray-conv-ppa-246									
12	perform adicio-nar-array-conv-ppa									

Fig. 9. The Test Cases from the First Experiment

5. Analyses and Discussions

The experiments performed by this work proved the relation between the source code cyclomatic complexity with the amount of test cases. Therefore, it was observed a significant productivity gain when automate the test cases generation.

Also, it was observed that scanning flow graph arcs is possible to produce a complete set of all possible paths. Indeed, when test cases have an automatic generation, it increases the possibility to detects defects at an early development stage.

According to Pressman [2], there is a strong relationship between the defect detection moment and the final product cost. The final cost trends to stabilize when a defect is found at an early development stage. Moreover, he says that the saving is up to a hundred times when compared to the final development stage.

Another important observed result is the code fragment visual representation trough the CFG. In this representation any team member is able to analyze the algorithm behavior. Also, it was providing malicious codes identification, like hidden IF statements.

6. Conclusion

This work has investigated the WbT utilization through CFT, within NATURAL source code fragments. As well as generating automatic from the analysed fragment its corresponding graph, and subsequently its respective test cases.

The proposed tool is based on Q1 and Q2, aiming to identify the possible paths in NATURAL algorithms. Moreover, it enables a friendly visualization of the algorithm paths.

The SERPRO provides necessary NATURAL codes fragments from its development environment, to validate this work. Stressing that the provided fragments have no confidential information and/or business aspects. The proposed tool was aimed to analysis its algorithm aspects only.

At the end, both carried out experiments have produced the expected result. That means, the CFG generation and respective test cases were generated for each fragment.

Based on experiment results, the proposed tool proved effective in the CFG creation and automated test cases generation. Thus, it provides the defects detection at early development stages, ensuring the software quality and reliability.

It is suggested to improve the grammar used in the syntactic analysis. It aims to allow the analysis of multiple condition and subroutine calls. Also, it is recommended add to this proposed tool a Graphical User Interface (GUI). It will decrease the complexity between users and the proposed tool usage.

Acknowledgment

The authors thank: to ITA, for its support during this work development; to SERPRO, for providing two NATURAL code fragments and allowing our work; and to 2RP Net enterprise for supporting some hardware infrastructure for this work.

References

[1] L. Molinari, *Testes de Software*: Produzindo Sistemas Melhores e Mais Confiáveis, 4th ed. São Paulo: Editora Érica, 2003.

[2] R. S. Pressman, *Software Engineering*: A Practitioner???s Approach, 7th ed. New York: McGraw-Hill, 2010.

[3] Software ag, *"NATURAL and Adabas Database."*. [Online]. Available: https://empower.softwareag.com/Products/. [Accessed: 10-Oct-2015].

[4] SERPRO, *"Brazilian Federal Service of Data Processing."* [Online]. Available: https://www.serpro.gov.br/home. [Accessed: 12-Aug-2015].

[5] Python Foundation, *"Python 2.7.10 documentation."* [Online]. Available: https://docs.python.org/2/. [Accessed: 06-Jun-2015].

[6] Computer Society of the IEEE, *"IEEE Standard Glossary of Software Engineering Terminology,"* New York, 1990.

[7] L. Crispin and J. Gregory, *Agile Testing*: A Practical Guide for Testers and Agile Teams, 1st ed. Crawfordsville: Addison-Wesley, 2009.

[8] G. Meszaros, *xUnit Test Patterns*: Refactoring Test Code, 1st ed. Boston: Addison-Wesley, 2007.

[9] K. Beck, *Extreme Programming Explained*: Embrace Change, 2nd ed. Boston: Addison-Wesley, 2005.

[10] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software*, 3rd ed. New Jersey: Wiley, 2011.

[11] L. Copeland, *A Practitioner???s Guide to Software Test Design*, 1st ed. Boston: Artech House, 2004.

[12] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*, Artech House, 1st ed. Boston: Artech House, 2002.

[13] C. W. Hetzel, *"The Complete Guide to Software Testing,"* Qual. Reliab. Eng. Int., vol. 2, no. 4, pp. 274???274, Oct. 1986.