MSc in Informatics Engineering
Dissertation
Intermediate Report

Evaluate the robustness of the Cloud

Gonçalo Silva Pereira
gsp@student.dei.uc.pt

Supervisor:
Raul Barbosa

Co-Supervisor:
Henrique Madeira

June 25, 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Dedication

Acknowledgements

I would like to thank Thomas Corbat and professors Raul Barbosa and Henrique Madeira, who are role models, by their support and help to make good decisions.

Thank my girlfriend for her support, understanding and the fellowship along this path. To my friends and colleagues of Department of Informatics Engineering for the patience and for all the times they have given me support.

Last but certainly not least, I would like to thank to my family for the encouragement, love and all the unconditional and constant support that let me fulfill this dream. Obrigado!

Gonçalo Silva Pereira

“ Bridges are normally built on-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always breaks down.

Alfred Z. Spector, Google Research ”

“ I have no special talents. I am only passionately curious.

Albert Einstein ”

Contents

List of Figures

List of Tables

Abstract

Nowadays, the Information and Communication Technologies are responsible for 2-4% of CO^2 emissions, but in the next five or ten years these will increase to 10% [1]. Because of this, the next challenge is to reduce the costs of ICT and its impact in the environment while the IC services keep growing.

Cloud computing is a new paradigm that provides on-demand self-service resources (computing, network and storage). It also promises to reduce the costs of ICT, but isn't free of external disturbance like security attacks, power surges, workload faults and others.

Therefore, the theme of my dissertation is "Evaluate the robustness of the Cloud". I will design and implement a fault injector for software coded in C to evaluate the capacity of the cloud to recover from faults.

Keywords: Faults, Errors, Failures, Vulnerabilities, Fault Injection, Fault Tolerance, Security, Robustness.

1 Introduction

1.1 Contextualization

The present dissertation describes the work developed in the scope of Master of Science in Informatics Engineering. It is focused on "Evaluate the robustness of Cloud" and this is a very important issue nowadays, because of the increasing usage of this. It's characterized by the placement of data and software on remote infrastructure. Despite the numerous benefits, the reliability of these platforms hasn't kept the needs, and users trust on their applications to systems outside of personal control.

In this context, the problem of confidence in the entity that manages the platform where applications have been executed arises naturally. Any organization that put an application in the cloud (for example, Microsoft Azure or Amazon EC2) so should accept the assurances given by the service provider.

1.2 The project

This project is based mainly in inject software faults. It was decided since there are already other people involved in the part of hardware faults.

1.3 Objectives

The main objective of this work is to evaluate the robustness of the cloud. To do that, I will design and implement a tool to inject software faults in source code of some applications.

Nevertheless, this objective is divided in some other goals:

- Generate derivations of main code of selected programs;
- Verify and analyze the effect of produced faults;
- Compile the programs with injected faults, by using make file.

1.4 Document Structure

In this document are specified all the related subjects with the project.

The second section presents the state-of-the-art in the related areas with particular emphasis to Cloud Computing and Fault Injection.

The third section is an important section of this report, because of the research involved in the execution of this work. It was necessary to take some important decisions based in research results, knowledge and my own experience.

The fourth section describes the work that has been done in Fault Injector, and the work that should be done in the next semester.

The fifth section explains other modules that need to be executed in this project to observe and evaluate the results of the fault injector.

In the last section, I will do an overview analyses of my work, in general the operators and the constraints developed. I will also talk about the work to be done in the next semester.

1.5 Management

1.5.1 Meetings

About the meetings, the supervisor Raul Barbosa and I agreed that meeting once every week was the best option. Moreover they happened, with one or another change of schedule to reconcile with the other activities from both. In addition, I attended some general meetings of the project. In them, we could discuss concepts and the direction of the project with colleagues and teachers, among them: Raul Barbosa (supervisor), Henrique Madeira (co-supervisor), João Durães and João André Ferro.

1.5.2 Risks

The main-risks of execution of this project are:

- Equipment Failure;
- Data lost;
- Publication of similar research;
- Personal issues interfering with the progress;
- Student loses interest;
- Dispute between student and supervisor;
- Supervisor takes excessive time to check final drafts;
- Student wants to submit thesis without supervisor approval.

The preventative measures and recovery measures can be seen at Appendix A.2 in other perspective.

1.5.3 Planning and Tracking

In Appendix A.1, is showed the Gantt diagram with the tasks that have been done during the first semester. As I postponed this dissertation for six months so, the scope and the context have changed. Now the two Gantt diagrams are incomparable.

2 State of the Art

Nowadays, people use many services based in the cloud and many companies choose to use them too. By doing that, companies reduce the costs of IT infrastructure and not even need to buy "physical storage", neither care where the data is. The cloud service provides that the data is secure. However, like any system, the cloud has problems such any other computer system, software and hardware faults. The resilience of the cloud is very important too. The increased use of cloud is related to a low usage of many dedicated servers, lower voltage levels, reduction of noise margins and increasing clock rates. The cloud provider offer resources ready to deliver [1].

There are many studies showing that the software faults [2] are the main cause of computer failures. But, the number of faults that can be emulated is directly related to the technique used.

	Software	Hardware
Hardware		HWIFI
Software	SWIFI	SWIFI

Table 1: Fault injection techniques and emulation environment.

- **Software Implemented Fault Injection (SWIFI)** - the goal of this technique is to emulate errors at software level that happen during the execution environment, in hardware or software; Examples: Data corruption in registers, memory or hard drive; Communication problems in network or NoC; Software faults in binary code, in object files or in source code.
- **Hardware Implemented Fault Injection (HWIFI)** - this technique is related to the fault injections in the final system hardware. Examples: Electromagnetic pulse (EMP), radiation.

SWIFI are an attractive technique because won't require additional hardware (increase the cost of test). The targets of this technique are the applications and the operating systems, but, this technique don't have only advantages, can't inject faults in inaccessible areas of software and may disrupt or change the workload of the testing software. This technique can be used at:

- **Compilation time at object code level** - Modify the structure of the program before the creation of executable file;
- **Execution environment at binary code level** - Changing the binary code activated by a timeout, an exception or a trap. At this level, less than seventy percent of the software faults can be emulated [3].
- **Before compile time at source code level** - Derivate the source code by removing, replacing or inserting some simple code before the compilation of program;

I had the opportunity to access to the application (executable only) of Robert Natella, named SAFE, that injects software faults, as I also intended to do (I will describe it in next section).

2.1 Software Implemented Fault Injection of Software Faults

JACA Tool

JACA[4] is a tool that has been made to validate Java applications. It injects high-level software faults and is based on computational reflection to inject interface faults in Java applications[5].

J-SWFIT

Java Software Fault Injection Tool[6] is a tool that doesn't need the source code to perform the injection, the mutation of the code is performed directly at byte-code level.

SAFE by Robert Natella

Safe is an application to inject realistic software faults in programs coded in C and C++. This tool uses MCPP as parser, to get the tree of code. The decision of using MCPP instead of GCC parser was a workaround for some of the shortcomings of the GCC's C preprocessor.

After that, write some files, variations of original files (code with simple mutations) with operators applied. Robert Natella implemented thirteen operators in SAFE, same as João Durães[7], but with the difference that Robert implemented at source code level, and João at binary level.

2.2 ODC Model

Orthogonal Defect Classification [8] Model is a framework developed by IBM [9], created to improve the level of technology available to assist the decisions of a software engineer, via measurement and analysis. ODC can be used to classify and analyze defects during software development.

For that, this model has eight categories:

- **Function** - This defect affects significant capability, end-user features, product Application Programming Interface, interface with hardware architecture, or global structure(s). It would require a formal design change.
- **Assignment** - Typically an assignment defect indicates an initialization of control blocks or a data structure.
- **Interface** - Problems in the interaction with other components, modules, device drivers, call statements, control blocks, or parameter lists.
- **Checking** - Based on the program logic that is checked and failed to validate data and values before the usage, loop conditions, etc.
- **Timing/serialization** - Errors that happen in shared and real-time resources.
- **Build/package/merge** - Errors that occur in the integration of library systems, management of changes, or in version control.
- **Documentation** - Errors in the documentation, that can be propagated to publications and maintenance notes.
- **Algorithm** - Problems that can be fixed by re-implementing an algorithm or local data structure, include efficiency or correctness that affect the task.

3 Research objectives and approach method

In this section are discussed the main aspects in study.

3.1 Cloud Computing

To understand a little more what the Cloud Computing means:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."[10].

Cloud Computing is a new way to delivery IT services on-demand (utility-oriented and Internet-centric). This services include all the computational power, from hardware infrastructure as a set of virtual machines to software services as development platforms and distributed applications.

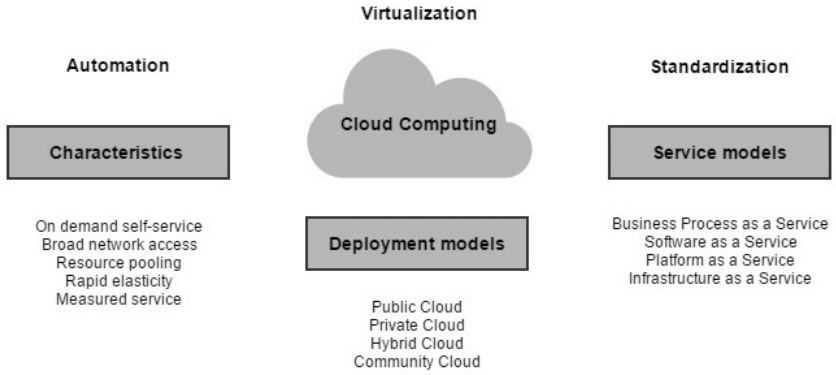

Figure 1: *Cloud computing overview.*

Below, I will describe it in relation to characteristics, deployment models and service models[11].

The characteristics of Cloud Computing are:

- **On demand self-service** - The users can request and manage their cloud computing resources without requiring human interaction, through a web-based self-service portal.
- **Broad network access** - Provide access over the network and using standard way through by several clients (e.g., mobile phones, tablets, laptops and workstations).
- **Resource pooling** - The computer resources are pooled to serve multiple customers through the safe separation of the resources at logical level.

- **Rapid elasticity** - Capability of resources to be elastically provisioned and released. Making sure that the application will have exactly the capacity that it needs at any point of time.
- **Measured service** - The service is monitored, measured, and reported transparently based on the usage. The clients pay in accordance with the service spent.

Four models of deployment:

- **Private Cloud** - It is a single-tenant cloud solution utilizing client hardware and software, is located inside the client firewall or even data center. The sensitive information is maintained inside of organization. It has the disadvantage of not having ability to scale on demand.
- **Community Cloud** - It is shared by organizations with similar interests, supported by a specific community, sharing the same mission, security requirements, etc.
- **Public Cloud** - It is available to the general public or to a group of a big company. It is a multi-tenant cloud solution owned by cloud service provider, that delivers shared hardware and software to clients private network (mostly the Internet) and data centers.
- **Hybrid Cloud** - Composed by two or more services (private, community or public), together by standard technologies or proprietary that allows portability. Takes advantages from the best of private and public. Example: A client can implement a private cloud for applications with sensitive data and a public cloud for other data, non-sensitive.

Four levels of Cloud Computing Service Models:

- **Infrastructure-as-a-Service** - As the name suggests, provides a computing infrastructure, such as virtual machines, firewalls, load balancers, IP addresses, virtual local area networks and others. Examples: Amazon EC2, Windows Azure.
- **Platform-as-a-Service** - Provides a computing platform, normally includes operating system, programming language execution environment, database, web server and others. Examples: AWS Elastic Beanstalk, Windows Azure, Heroku.
- **Software-as-a-Service** - Provides access to application softwares often referred as *on-demand self-service* software. Use it without install, setup, and run the application. Service provider does all those things for you. Examples: Google Apps, Microsoft Office 365.

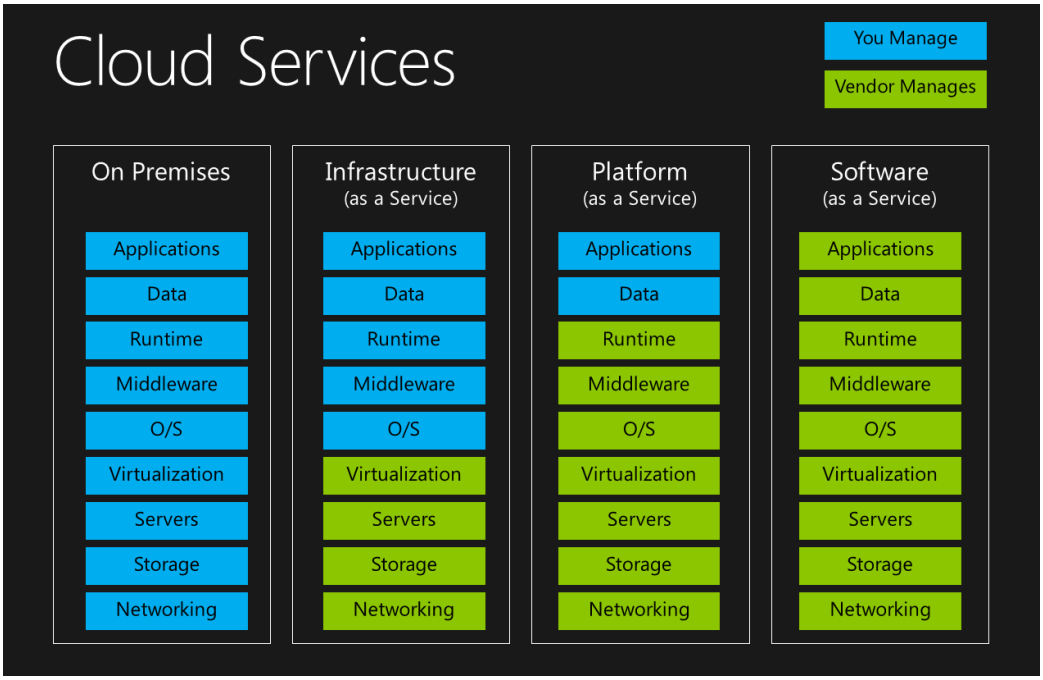

Figure 2: *Cloud computing service models.*

- **Business-Process-as-a-Service** - This model provides an entire horizontal or vertical business process and builds on top of any of services previously described.

Nevertheless, such as any computer system, cloud computing isn't free of external disturbances[1], the most important are:

- Security attacks;
- Accidents;
- Power surges;
- Workload faults;
- Malfunction;
- Worms;
- Distributed Denial of Service attacks.

3.2 Tools - GCC Parser, Bison and Eclipse CDT

In the beginning of planning the basic software without any user interface, it was necessary to research the best applications, as the best way for using them to obtain panned results (fault injector). For that, I thought that I could use the same tools that I have used in Compilers course, Lex and Yacc.

GCC Parser

Nowadays, GCC use a hand-written parser to improve syntactic error diagnostics, giving people meaningful messages on syntax errors.

Eclipse CDT

Eclipse CDT, as the name suggests, is a plugin for Eclipse that provides a fully functional C and C++ Integrated Development Environment. Some of the features included in this plugin that are interesting for this project are:

- Source navigation;
- Code editor with syntax highlighting;
- Source code refactoring and code generation.

It's possible to use this plugin in standalone mode, importing .jar files to the project. Using it, I can code Fault Injector in Java, making the software more maintainable and easy to use, write, compile and debug.

In the end, I selected Eclipse CDT Plugin as standalone (only import libraries to project), because of my abilities in programming in Java Language, the maintainability of software, the low learning level than the developers need to modify it.

4 Fault Injector Development

The Fault Injector currently in development is coded in Java using Eclipse CDT, and it will have thirteen operators (can be seen in Table 2)[12].

Fault Type	Description
MFC	Missing function call
MVIV	Missing variable initialization with a value
MVAV	Missing variable assignment with a value
MVAE	Missing variable assignment with an expression
MIA	Missing if construct around statements
MIFS	Missing if construct and surrounded statements
MIEB	Missing if construct plus statements plus else before statements
MLAC	Missing and sub-expr. in logical expression used in branch condition
MLOC	Missing or sub-expr. in logical expression used in branch condition
MLPA	Missing localized part of the algorithm
WVAV	Wrong value assigned to a variable
WPFV	Wrong variable used in parameter of function call
WAEP	Wrong arithmetic expression in parameters of function call

Table 2: *Fault emulation operators.*

In the beginning of this project,

4.1 Generate derivations

I chose to use a set of the most representative faults, previously specified by João Durães[7], specified individually further down:

4.1.1 MIFS

- Missing if construct and surrounded statements - **Implemented**

The application of this operator changes the source code with the remotion of one *if* construct and the statements surrounded by it. But, to do that, I need to verify the constraints above:

- **C02** - Call must **not be** the only statement in the block;
- **C08** - The if construct must **not be** associated to an else construct;
- **C09** - Statements must **not include** more than five statements and not include loops.

4.1.2 MLAC

- Missing and sub-expr. in logical expression used in branch condition - **Implemented**

This operator emulates the remotion of part of a logical expression used in a branch condition. To apply this operator, the code must have at least two branch conditions linked together with the logical operator AND. With an AND operator, if one of the sub-expressions is *false* all the expression will be *false* and the condition will fail.

- **C12** - Must have **at least two** branch conditions.

4.1.3 MFC

- Missing function call

The emulation of this operator is based in the remotion of a function call in a context where the returned value is not used.

- **C01** - Return value of the function must **not** be used;
- **C02** - Call must **not be** the only statement in the block.

4.1.4 MIA

- Missing if construct around statements - **Implemented**

- **C08** - The if construct must **not be** associated to an else construct;
- **C09** - Statements must **not include** more than five statements and not include loops.

4.1.5 MLOC

- Missing or sub-expr. in logical expression used in branch condition - **Implemented**

This operator emules the remotion of part of a logical expression used in a branch condition. To apply this operator, the code must have at least two branch conditions linked together with the logical operator OR.

- **C12** - Must have **at least two** branch conditions.

4.1.6 MLPA

- Missing localized part of the algorithm

- **C02** - Call must **not be** the only statement in the block;
- **C10** - Statements are in the same block, **do not include** more than 5 stats, or loops.

4.1.7 MVAE

- Missing variable assignment with an expression

- **C02** - Call must **not be** the only statement in the block;
- **C03** - Variable must **be** inside stack frame;
- **C06** - Assignment must **not be** part of a for construct;
- **C07** - Must **not be** the first assignment for that variable in the module.

4.1.8 MVAV

- Missing variable assignment with a value

- **C02** - Call must **not be** the only statement in the block;
- **C03** - Variable must **be** inside stack frame;
- **C06** - Assignment must **not be** part of a for construct;
- **C07** - Must **not be** the first assignment for that variable in the module.

4.1.9 MIEB

- Missing if construct plus statements plus else before statements - **Implemented**

This operator generates derivations of the source code of applications by removing the if construct plus statements plus else before statements. To apply this operator I need to verify the constraint above:

- **C08n** - The if construct must **be** associated to an else construct.

This constraint does not exists in João Durães specification, but as this operator cannot be applied in all situations, I implemented and specify it.

4.1.10 MVIV

- Missing variable initialization with a value

- **C02** - Call must **not be** the only statement in the block;
- **C03** - Variable must **be** inside stack frame;
- **C04** - Must **be** the first assignment for that variable in the module;
- **C05** - Assignment must **not be** inside a loop;
- **C06** - Assignment must **not be** part of a for construct.

4.1.11 WVAV

- Wrong value assigned to a variable

- **C03** - Variable must **be** inside stack frame;
- **C04** - Must **be** the first assignment for that variable in the module;
- **C06** - Assignment must **not be** part of a for construct.

4.1.12 WAEP

- Wrong arithmetic expression in parameters of function call

4.1.13 WPFV

- Wrong variable used in parameter of function call

- **C03** - Variable must **be** inside stack frame;
- **C11** - There must **be at least** two variables in this module.

The operators above will be applied to source code of applications and will generate modified files.

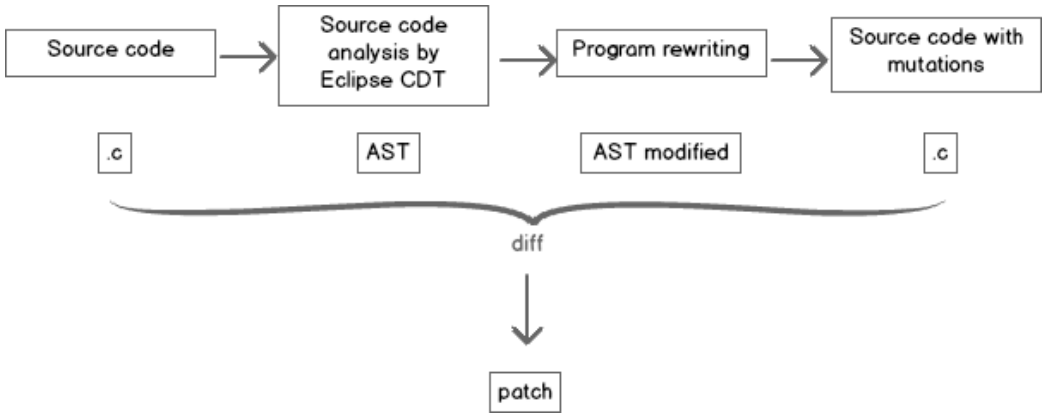

Figure 3: *Overview of the injection tool.*

4.2 Constraints

The constraints defined below were specified by João Durães in ... .

Constraints	Description
C01	Return value of the function must **not** be used
C02	Call must **not be** the only statement in the block
C03	Variable must **be** inside stack frame
C04	Must **be** the first assignment for that variable in the module
C05	Assignment must **not be** inside a loop
C06	Assignment must **not be** part of a for construct
C07	Must **not be** the first assignment for that variable in the module
C08	The if construct must **not be** associated to an else construct
C09	Statements must **not include** more than five statements and not include loops
C10	Statements are in the same block, **do not include** more than 5 stats, or loops
C11	There must **be at least** two variables in this module

Table 3: *Fault emulation contraints defined by João Durães.*

Constraints	Description
C08n	The if construct must **be** associated to an else construct
C12	Must have **at least two** branch conditions

Table 4: *Other constraints.*

4.3 Applications to inject faults

5 Work plan and implications

Built three separated modules:

- Generate the derivations of main code of selected programs;
- Verify and analyze the effect of produced faults;
- Compile the programs with injected faults, by using make file.

5.1 Compile programs

5.2 Analyze the effects

After the compilation and execution of the programs, the results need to be evaluated. To measure that, I will use the *Koopman's CRASH Scale*[13]:

- **C**atastrophic - Operating System crashed or multiple tasks affected;
- **R**estart - Task or process hangs, requiring restart;
- **A**bort - Task or process aborts abnormally (i.e. "code dump" or "segmentation violation");
- **S**ilent - Test Process exits without an error code returned when one should exist;
- **H**indering - Test Process exits with an error code not relevant to the situation or incorrect error code returned;
- **P**ass - The module exits properly, possibly with an appropriate error code.

This *CRASH Scale* is one way to show results of the effect of faults on an end-use system, mainly from the operating system perspective.

6 Conclusion

6.1 Global Vision

In table 5, it's possible to understand the operators that were implemented in the first semester of this dissertation. As can be seen, I have implemented five of thirteen operators that João Durães specified.

In table 6, is also possible to check that I have implemented three of eleven constraints related to the thirteen operators.

MISSING	MIFS	Missing IF construct and surrounded Statements	C02 C08 C09
	MLAC	Missing "and sub-expression" in logical expression used in branch condition	C12
	MFC	**Missing function call**	C01 C02
	MIA	Missing IF Around statements	C08 C09
	MLOC	Missing "or sub-expression" in logical expression used in branch condition	C12
	MLPA	Missing Localized Part of the Algorithm	C02 C10
	MVAE	Missing Variable Assignment with an Expression	C02 C03 C07 C06
	~~MFCT~~		
	MVAV	Missing Variable Assignment with a Value	C02 C03 C07 C06
	MIEB	Missing IF construct plus statements plus else before statements	C08n
	MVIV	Missing Variable initialization with a value	C02 C03 C04 C05 C06
WRONG	~~WLEC~~		
	~~WALL~~		
	WVAV	Wrong Value Assigned to a Variable	C03 C04 C06
	WAEP	Wrong Arithmetic Expression in a function Parameter	
	~~WSUT~~		
	WPFV	Wrong Variable in parameter of function Call	C03 C11
Extraneous	~~EVAV~~		

Implementado
Em vista
Em falta

Table 5: *State of the operators and its constraints.*

Current	C01	Return value of the function **must not** being used
	C02	Call **must not be** the only statement in the block
	C03	Variable **must be** inside stack frame
	C04	**Must be** the first assignment for that variable in the module
	C05	Assignment **must not be** inside a loop
	C06	Assignment **must not be** part of a for construct
	C07	**Must not be** the first assignment for that variable in the module
	C08	The if construct **must not be** associated to an else construct
	C09	Statements **must not include** more than five statemens and not include loops
	C10	Statements are in the same block, **do not include** more than 5 stats. or loops
	C11	There **must be** at leat two variables in this module

		Operators	Versions
Extra	C08n	The if construct **must be** associated to an else construct	**MIEB** a
	C12	Must have **at least two** branch conditions	**MLAC** **MLOC** b
			c
			d
			e
			f
			g
			h

Implementado
Em vista

Table 6: *State of the constraints.*

6.2 Future Work

In the future, I have planned to implement the other operators and constraints. In addition, apply this software in testing of open source software's that I will select.

I will use regression testing to verify if when I coded one new operator or constraint I didn't mess with the operators and constraints previous implemented. From version to version, I use a regression testing to test the fault injector to guarantee that application doesn't regarded.

"The purpose of regression testing is to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change. Regression testing is usually performed by running some, or all, of the test cases created to test modifications in previous versions of the software."

A Appendix

A.1 Appendix A - Gantt diagrams

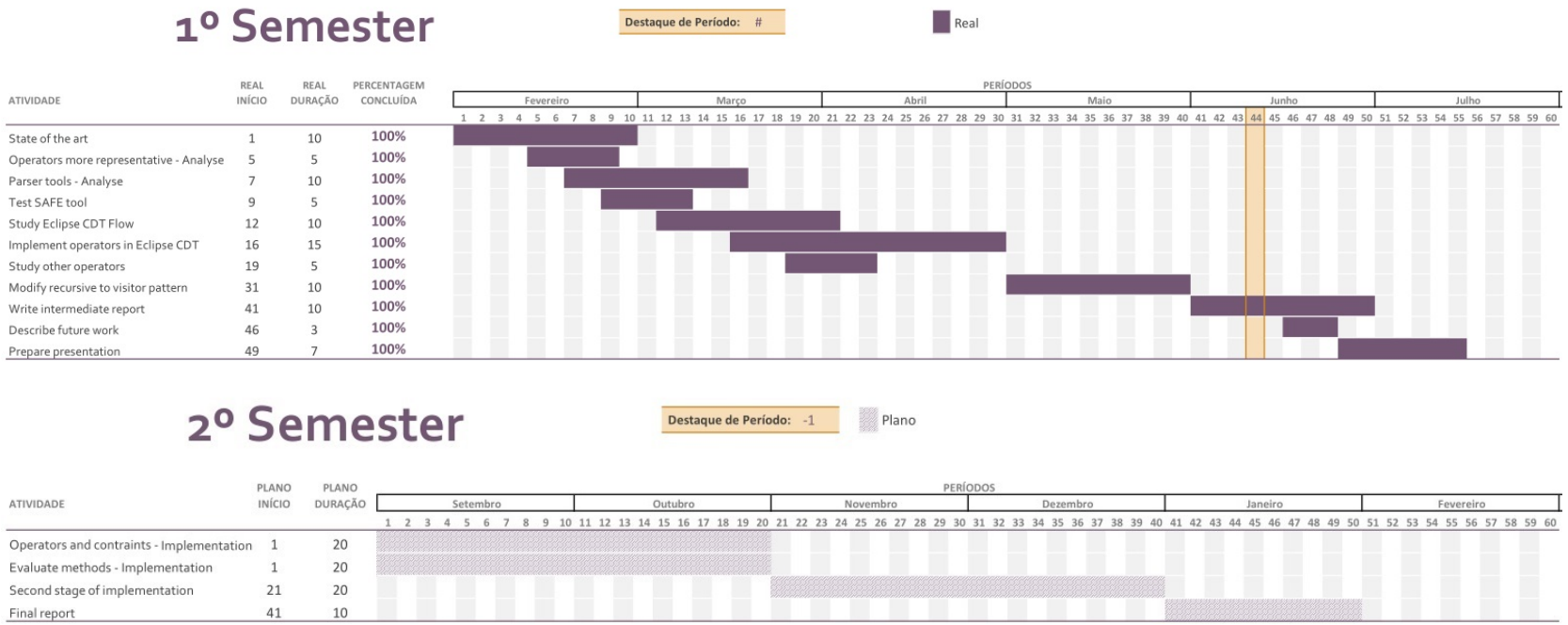

Figure 4: *First and second semester gantt.*

A.2 Appendix B - Risks table

Risc Area	Preventative Measures	Recovery Measures
Equipment Failure	Ensure regular maintenance is undertaken	Use alternative sources/type of equipment as appropriate
	Allow for sufficient funding for repairs	
	Indentify alternative sources/type of equipment	
Data lost	Back-up data regularly	
Publication of similar research	Regularly search electronic publications databases	Modify project
	Continue literature review throughout candidature	
	Ensure timely submission	
Personal issues interfere with progress	Take leave of absence (unless for sickness or bereavement)	Re-apply for admission when able to commit
	Take annual leave	
	Take sick leave	
	Communicate with supervisor	
Student loses interest	Select motivating topic at the start	
	Enrolling area ensures a dynamic research culture	
	Improve communication between student and supervisor	
	Look for warning signs	
	Register for support programs/seminars	
	Talk to fellow students in research area	
Dispute between student and supervisor	Understand each other's roles and expectations	
	Agree on dispute resolution process when initiating relationship	
Supervisor takes excessive time to check final drafts	Supervisor to plan out workload	
	Student plan ahead to ensure supervisor will be available	
	Student/Supervisor to review chapters/sections at regular intervals	
Student wants to submit thesis without supervisor approval	Student to be counselled regarding implications - a recomendation of fail or major revision from examiners likely if thesis below standard	Review of thesis by alternative person within University recommended

Figure 5: *Risks.*

A.3 Appendix C - Decision tree

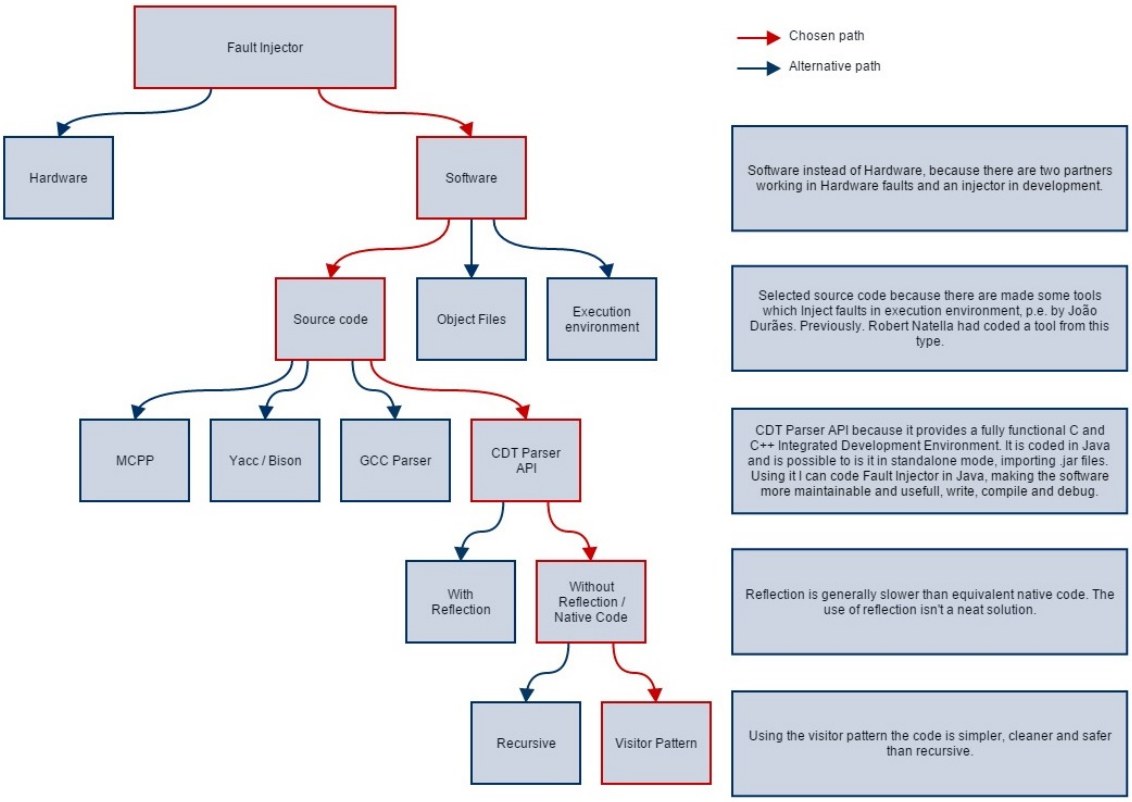

Figure 6: *Decision tree.*

A.4 Appendix D - Abbreviations

API Application Programming Interface

BPaaS Business-Process-as-a-Service

DDOS Distributed Denial of Service

EMP Electromagnetic pulse

HWIFI Hardware Implemented Fault Injection

IaaS Infrastructure-as-a-Service

ODC Orthogonal Defect Classification

PaaS Platform-as-a-Service

SaaS Software-as-a-Service

SWIFI Software Implemented Fault Injection

Operators

MFC missing function call

MIA missing if construct around statements

MIEB missing if construct plus statements plus else before statements

MIFS missing if construct and surrounded statements

MLAC missing and sub-expr. in logical expression used in branch condition

MLOC missing or sub-expr. in logical expression used in branch condition

MLPA missing localized part of the algorithm

MVAE missing variable assignment with an expression

MVAV missing variable assignment with a value

WAEP wrong arithmetic expression in parameters of function call

WPFV wrong variable used in parameter of function call

WVAV wrong value assigned to a variable

Constraints

C01 return value of the function must **not** be used

C02 call must **not be** the only statement in the block

C03 variable must **be** inside stack frame

C04 must **be** the first assignment for that variable in the module

C05 assignment must **not be** inside a loop

C06 assignment must **not be** part of a for construct

C07 must **not be** the first assignment for that variable in the module

C08 the if construct must **not be** associated to an else construct

C08n the if construct must **be** associated to an else construct

C09 statements must **not include** more than five statements and not include loops

C10 statements are in the same block, **do not include** more than 5 stats, or loops

C11 there must **be at least** two variables in this module

C12 must have **at least two** branch conditions

References

[1] K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, *Resilience assessment and evaluation of computing systems.* Springer, 2012.

[2] A. Avizzienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing."

[3] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on.* IEEE, 2000, pp. 417–426.

[4] L. Regina, E. Martins *et al.*, "Jaca—a software fault injection tool," in *null.* IEEE, 2003, p. 667.

[5] E. Martins, C. M. Rubira, and N. G. Leme, "Jaca: A reflective fault injection tool based on patterns," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, 2002, pp. 483–487.

[6] B. P. Sanches, T. Basso, and R. Moraes, "J-swfit: A java software fault injection tool," in *Dependable Computing (LADC), 2011 5th Latin-American Symposium on.* IEEE, 2011, pp. 106–115.

[7] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 849–867, 2006.

[8] N. Bridge and C. Miller, "Orthogonal defect classification using defect data to improve software development," *Software Quality*, vol. 3, no. 1, pp. 1–8, 1998.

[9] R. Chillarege, *Orthogonal Defect Classification.* Handbook of Software Reliability Engineering, ed. Michael R. Lyu (Los Alamitos, CA: IEEE Computer Science Press, 2004.

[10] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.

[11] E. Schouten, *IBM® SmartCloud® Essentials.* Packt Publishing Ltd, 2013.

[12] J. A. Duraes, "Faultloads baseadas em falhas de software para testes padronizados de confiabilidade," *Thesis*, pp. 0–269, 2005.

[13] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on.* IEEE, 1997, pp. 72–79.