



MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO  
**INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS**

sid.inpe.br/mtc-m21b/2016/04.20.19.30-TDI

**REACTOR: COMBINING STATIC ANALYSIS, TESTING  
AND REVERSE ENGINEERING TO DETECT  
SOFTWARE DEFECTS**

Alessandro Oliveira Arantes

Doctorate Thesis of the Graduate  
Course in Applied Computing,  
guided by Drs. Valdivino  
Alexandre de Santiago Júnior,  
and Nandamudi Lankalapalli  
Vijaykumar, approved in may 02,  
2016.

URL of the original document:

<<http://urlib.net/8JMKD3MGP3W34P/3LHFRE2>>

INPE  
São José dos Campos  
2016

**PUBLISHED BY:**

Instituto Nacional de Pesquisas Espaciais - INPE

Gabinete do Diretor (GB)

Serviço de Informação e Documentação (SID)

Caixa Postal 515 - CEP 12.245-970

São José dos Campos - SP - Brasil

Tel.:(012) 3208-6923/6921

Fax: (012) 3208-6919

E-mail: pubtc@inpe.br

**COMMISSION OF BOARD OF PUBLISHING AND PRESERVATION  
OF INPE INTELLECTUAL PRODUCTION (DE/DIR-544):****Chairperson:**

Maria do Carmo de Andrade Nono - Conselho de Pós-Graduação (CPG)

**Members:**

Dr. Plínio Carlos Alvalá - Centro de Ciência do Sistema Terrestre (CST)

Dr. André de Castro Milone - Coordenação de Ciências Espaciais e Atmosféricas  
(CEA)

Dra. Carina de Barros Melo - Coordenação de Laboratórios Associados (CTE)

Dr. Evandro Marconi Rocco - Coordenação de Engenharia e Tecnologia Espacial  
(ETE)

Dr. Hermann Johann Heinrich Kux - Coordenação de Observação da Terra (OBT)

Dr. Marley Cavalcante de Lima Moscati - Centro de Previsão de Tempo e Estudos  
Climáticos (CPT)

Silvia Castro Marcelino - Serviço de Informação e Documentação (SID) **DIGITAL**

**LIBRARY:**

Dr. Gerald Jean Francis Banon

Clayton Martins Pereira - Serviço de Informação e Documentação (SID)

**DOCUMENT REVIEW:**

Simone Angélica Del Duca Barbedo - Serviço de Informação e Documentação  
(SID)

Yolanda Ribeiro da Silva Souza - Serviço de Informação e Documentação (SID)

**ELECTRONIC EDITING:**

Marcelo de Castro Pazos - Serviço de Informação e Documentação (SID)

André Luis Dias Fernandes - Serviço de Informação e Documentação (SID)



MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO  
**INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS**

sid.inpe.br/mtc-m21b/2016/04.20.19.30-TDI

**REACTOR: COMBINING STATIC ANALYSIS, TESTING  
AND REVERSE ENGINEERING TO DETECT  
SOFTWARE DEFECTS**

Alessandro Oliveira Arantes

Doctorate Thesis of the Graduate Course in Applied Computing, guided by Drs. Valdivino Alexandre de Santiago Júnior, and Nandamudi Lankalapalli Vijaykumar, approved in may 02, 2016.

URL of the original document:

<<http://urlib.net/8JMKD3MGP3W34P/3LHFRE2>>

INPE  
São José dos Campos  
2016

Cataloging in Publication Data

---

Arantes, Alessandro Oliveira.

Ar14r REACTOR: combining static analysis, testing and reverse engineering to detect software defects / Alessandro Oliveira Arantes. – São José dos Campos : INPE, 2016.  
xxiv + 186 p. ; (sid.inpe.br/mtc-m21b/2016/04.20.19.30-TDI)

Thesis (Doctorate in Applied Computing) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2016.

Guiding : Drs. Valdivino Alexandre de Santiago Júnior, and Nandamudi Lankalapalli Vijaykumar.

1. Static code analysis. 2. Software testing. 3. Reverse engineering. 4. Test case generation. 5. Test oracle. I.Title.

CDU 004.415:005.59

---



Esta obra foi licenciada sob uma Licença [Creative Commons Atribuição-NãoComercial 3.0 Não Adaptada](https://creativecommons.org/licenses/by-nc/3.0/).

This work is licensed under a [Creative Commons Attribution-NonCommercial 3.0 Unported License](https://creativecommons.org/licenses/by-nc/3.0/).

Aluno (a): **Alessandro Oliveira Arantes**

Título: "REACTOR: COMBINING STATIC ANALYSIS, TESTING AND REVERSE ENGINEERING TO DETECT SOFTWARE DEFECTS".

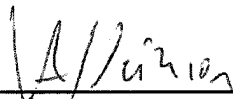
Aprovado (a) pela Banca Examinadora  
em cumprimento ao requisito exigido para  
obtenção do Título de **Doutor(a)** em  
**Computação Aplicada**

Dr. Solon Venâncio de Carvalho



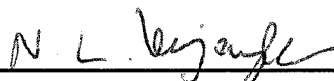
Presidente / INPE / SJC Campos - SP

Dr. Valdivino Alexandre de Santiago Júnior



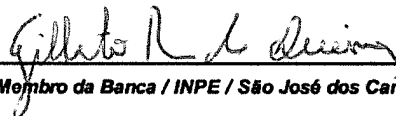
Orientador(a) / INPE / São José dos Campos - SP

Dr. Nandamudi Lankalapalli Vijaykumar



Orientador(a) / INPE / SJC Campos - SP

Dr. Gilberto Ribeiro de Queiroz



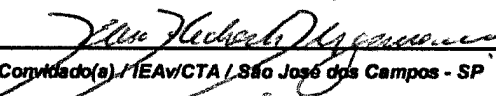
Membro da Banca / INPE / São José dos Campos - SP

Dra. Daniela Leal Musa



Convidado(a) / UNIFESP / São José dos Campos - SP

Dr. Elcio Hideiti Shiguemori



Convidado(a) / IEAv/CTA / São José dos Campos - SP

Este trabalho foi aprovado por:

maioria simples

unanimidade



*“I tell you folks, it’s harder than it looks.  
It’s a long way to the top if you wanna rock ‘n’ roll”.*

AC/DC  
em “*High Voltage*”, 1976





*A meus pais **Ailton e Maria**, e a minha esposa **Érica***



## ACKNOWLEDGEMENTS

Primeiramente, o maior dos agradecimentos aos orientadores Valdivino Santiago e Nandamudi Vijaykumar pela minha aceitação como aluno do curso e pela disposição, paciência e dedicação na condução deste trabalho desde a sugestão da proposta inicial até às versões finais.

Gostaria de agradecer também aos colegas do INPE que me acompanharam e compartilharam “sofrimentos” nesses últimos anos: Érica Souza, Luciana Brasil, Marlon da Silva, Rogério Marinke e muitos outros cujos nomes tornariam esta lista interminável.

Preciso também fazer um agradecimento ao Sherfis Ruwer, pelo excelente curso de Java que foi indispensável para a implementação deste trabalho; ao Rogério Marinke, cuja pesquisa feita em sua dissertação de mestrado contribuiu muito para o desenvolvimento da ferramenta; e ao Paulo Paiva, pelo curso essencial de expressões regulares.

Agradeço também a todos os colegas do IEAv, em especial ao Remo Carnevalli e Walter Magalhães, que permitiram minha dedicação ao curso dando-me um apoio provavelmente até maior do que o merecido. À Mônica De Marchi e Maria José que me ajudaram a conseguir recursos para publicar. À Maria Helena que me ajudou muito a resolver tudo que foi preciso no RH.

Da mesma forma, fico muito grato a meus pais Ailton e Maria Clara, cujo esforço de vida me possibilitou o estudo e o desenvolvimento profissional e pessoal.

É impossível não agradecer a minha esposa Érica Monquero por todo apoio, e principalmente, pela paciência em relevar minhas noites de trabalho em casa.

Também, aos familiares e amigos que sempre torceram incondicionalmente por mim e sempre se comoveram com meu “sofrimento”, deixo meus sinceros agradecimentos.

Ficarei eternamente grato também a todos aqueles que de alguma forma contribuíram com o trabalho, ou mesmo aos que somente contribuíram para que minha passagem pelo curso fosse inesquecível como foi.

E por último, mas não menos importante, agradeço a Deus por ter me dado saúde, paz e disposição para que eu pudesse me dedicar ao curso e atingir esse importante marco em minha vida.



## ABSTRACT

It is increasingly common the use of computer systems to replace human labor in critical systems, and since these systems have become more autonomous in decision making, they demand a high degree of quality and robustness. INPE develops embedded systems for scientific satellites and stratospheric balloons; consequently, the process of verification and validation require special care in detecting and preventing defects. In terms of complexity and system's domain in question, these processes consume specialist's manpower for a long period. In this scenario, the application of techniques that can automatically support test process provide a significant gain in specialist's productivity and efficiency. For this purpose, this work performs the source code reverse engineering in order to support a combination of two V&V processes, static source code analysis and software testing, in order to detect a wider range of defects. The proposed method, called REACTOR (Reverse Engineering for stAtic Code analysis and Testing to detect sOftwaRe defects), complements the traditional way that static code analyzers work by using dynamic information obtained by an automated test case generator, which combines three different black box techniques, being also possible to infer a set of estimated expected results similar to a test oracle. However, the combination of such techniques is not trivial, especially in terms of tasks that commonly demand some action that are not easily automated. Furthermore, the static analysis by itself can not reveal several types of defects that can only be detected by combining static analysis and dynamic information. The REACTOR method has been implemented in a software tool, also called REACTOR, which exempts from a large manual labor's amount from testers by automating the process and basing only on application's source code. In addition, REACTOR was applied to some case studies including one of the space application domain, and it performed better than three other well known static code analyzers.

Keywords: Static Code Analysis. Software Testing. Reverse Engineering. Test Case Generation. Test Oracle.



# REACTOR: COMBINANDO ANÁLISE ESTÁTICA, TESTE DE SOFTWARE E ENGENHARIA REVERSA PARA DETECÇÃO DE DEFEITOS DE SOFTWARE

## RESUMO

É cada vez mais comum a utilização de sistemas computacionais em substituição à mão de obra humana em sistemas críticos, e na medida em que estes sistemas têm se tornado mais autônomos para tomar decisões, eles exigem um alto grau de qualidade e robustez. O INPE desenvolve sistemas embarcados para satélites científicos e balões estratosféricos; conseqüentemente, os processos de verificação e validação exigem cuidados especiais na detecção e prevenção de defeitos. E tendo em vista a complexidade e o domínio dos sistemas em questão, estes processos consomem a mão de obra especialista por um longo período. Neste cenário, a aplicação de técnicas que possam efetuar testes de forma automática auxiliam o processo proporcionando um ganho significativo de produtividade e eficácia no trabalho dos especialistas. Com esse objetivo, este trabalho realiza a engenharia reversa de código-fonte de modo a combinar dois processos de V&V, análise estática de código fonte e teste de software, a fim de detectar uma gama mais ampla de defeitos. O método proposto, denominado REACTOR (Reverse Engineering for stAtic Code analysis and Testing to detect sOftwaRe defects), complementa a maneira tradicional pela qual os analisadores de código estático trabalham usando informações dinâmicas obtidas por um gerador de caso de teste automatizado, que combina três técnicas de caixa preta diferentes, sendo também possível inferir um conjunto de resultados esperados estimados similar a um oráculo de teste. Ainda assim, a leitura do código fonte estático por si só pode não revelar vários tipos de defeitos que só podem ser detectados combinando a análise estática com informação dinâmica. O método REACTOR foi implementado em uma ferramenta de software, também chamado de REACTOR, que poupa os testadores de um grande volume de trabalho manual automatizando o processo e baseando-se apenas no código fonte. Além disso, a REACTOR foi aplicada em alguns casos de estudo incluindo uma aplicação da área espacial, e seu desempenho foi melhor do que outras três conhecidos analisadores de código estático.

Palavras-chave: Análise Estática de Código Fonte. Testes de Software. Engenharia Reversa. Geração de Caso de Teste. Oráculo de Teste.





## LIST OF FIGURES

	<u>Page</u>
1.1 JUnit test case script. . . . .	5
2.1 Defect, Error and Failure Terminology. . . . .	9
2.2 Test Process. . . . .	17
2.3 Example of an BVA application. . . . .	20
2.4 Example of an EP application. . . . .	22
2.5 Usual control structures illustrated as CFG. . . . .	24
2.6 The basic structure of a test oracle. . . . .	27
3.1 The REACTOR method workflow. . . . .	33
3.2 Source code of <i>factorial problem</i> . . . . .	34
3.3 Input and output of <i>factorial problem</i> source code. . . . .	35
3.4 Automatic Equivalent and Boundary Partition Analysis for integer type variables. . . . .	36
3.5 Scopes annotated in <i>factorial problem</i> source code. . . . .	48
3.6 Control structures annotated in <i>factorial problem</i> source code. . . . .	49
3.7 Structural model of the SUVV created by REACTOR. . . . .	51
3.8 Example of source code decomposition. . . . .	52
3.9 Example of source code decomposition of <i>factorial problem</i> . . . . .	55
3.10 Code lines of <i>factorial problem</i> after the decomposition process. . . . .	56
3.11 Code lines of <i>factorial problem</i> arranged in code blocks. . . . .	59
3.12 REACTOR: distribution of the 47 types of static defects into classes. . . . .	73
3.13 Defects detected in <i>factorial problem</i> . . . . .	83
4.1 REACTOR's sequence diagram. . . . .	89
4.2 REACTOR's package diagram. . . . .	90
4.3 Configuration class of <i>factorial problem</i> . . . . .	92
4.4 Configuration class of <i>triangle classification</i> . . . . .	92
4.5 Source code decomposition and classification. . . . .	96
4.6 Conversion of source code into CFG. . . . .	97
4.7 Temporary variables created by REACTOR in <i>factorial problem</i> . . . . .	98
4.8 Text file with input and output data of <i>factorial problem</i> . . . . .	99
4.9 Classes of defects found in <i>factorial problem</i> . . . . .	100
4.10 List of defects found in <i>factorial problem</i> . . . . .	101
4.11 Running the SUVV with test cases manually. . . . .	107
4.12 Running the oracle procedure manually. . . . .	108

4.13	Test oracle generated for in <i>factorial problem</i> . . . . .	109
5.1	REACTOR's true positive static defects detection compared with other tools (Part I). . . . .	113
5.2	REACTOR's true positive static defects detection compared with other tools (Part II). . . . .	114
5.3	REACTOR's testing defects detection. . . . .	115
5.4	Rate of true positive static defects. . . . .	121
5.5	Rate of unimportant defects. . . . .	122

## LIST OF TABLES

	<u>Page</u>
3.1 Test cases generated for the <i>factorial problem</i> . . . . .	37
3.2 Example of a test case suite exhaustively generated. . . . .	40
3.3 Example of a test case suite reduced by pairwise. . . . .	41
3.4 Reduction rate of pairwise test cases. . . . .	41
3.5 Full code line <i>regex</i> used in REACTOR. . . . .	45
3.6 Inline code <i>regex</i> used in REACTOR. . . . .	46
3.7 Annotations to Delimit Scopes, Packages and Imports. . . . .	48
3.8 Annotations Identify Control Structures. . . . .	50
3.9 Annotations to Classify Code Lines. . . . .	57
3.10 Code block of <i>factor</i> in REACTOR. . . . .	60
3.11 Process used in <i>factorial problem</i> . . . . .	61
3.12 Scopes used in <i>factorial problem</i> . . . . .	62
3.13 Running example of <i>factorial problem</i> . . . . .	69
3.14 Last values set in <i>factorial problem</i> . . . . .	71
4.1 Suggested limits for testing boolean variables (1 bit). . . . .	93
4.2 Suggested limits for testing char variables (2 bytes/16 bits). . . . .	93
4.3 Suggested limits for testing byte variables (1 byte/8 bits). . . . .	94
4.4 Suggested limits for testing short variables (2 bytes/16 bits). . . . .	94
4.5 Suggested limits for testing integer variables (4 bytes/32 bits). . . . .	94
4.6 Suggested limits for testing long variables (8 bytes/64 bits). . . . .	95
4.7 Suggested limits for testing float variables (4 bytes/32 bits). . . . .	95
4.8 Suggested limits for testing double variables (8 bytes/64 bits). . . . .	95
4.9 Defects detected by REACTOR. . . . .	102
A.1 Full code line <i>regex</i> used in REACTOR. . . . .	138
A.2 Inline code <i>regex</i> used in REACTOR. . . . .	147
A.3 Set of standard lines for source code classification. . . . .	151



## LIST OF ABBREVIATIONS

AEBPA	– Automatic Equivalent and Boundary Partition Analysis
AI	– Artificial Intelligence
ANN	– Artificial Neural Networks
BVA	– Boundary Values Analysis
CFG	– Control Flow Graph
DCTA	– <i>Departamento de Ciência e Tecnologia Aeroespacial</i>
DFD	– Data Flow Diagram
EGSE	– Electrical Ground Support Equipment
EP	– Equivalence Partitioning
ERD	– Entity-Relationship Diagram
FSM	– Finite-State Machine
GUI	– Graphical User Interface
HTML	– HyperText Markup Language
IDE	– Integrated Development Environment
INPE	– <i>Instituto Nacional de Pesquisas Espaciais</i>
InTOL	– Intelligent Test Oracle Library
JDK	– Java Development Kit
JML	– Java Modeling Language
LFA	– Log File Analysis
MBT	– Model-Based Testing
ML	– Machine Learning
NL	– Natural Language
ORCAS	– <i>Observação de Raios Cósmicos Anômalos e Solares na Magnetosfera</i>
PHP	– Hypertext Preprocessor
REACTOR	– Reverse Engineering for stAtic Code analysis and Testing to detect sOftwaRe defects
RT	– Random Testing
SDL	– Specification and Description Language
SLA	– Supervised Learning Algorithm
SQL	– Structured Query Language
SSCARE	– Static Source Code Analysis by Reverse Engineering
SUVV	– Software Under Verification and Validation
TOG	– Test Oracle Generator
UML	– Unified Modeling Language
V&V	– Verification and Validation
VB	– Visual Basic
XML	– eXtensible Markup Language



# CONTENTS

	<u>Page</u>
<b>1 INTRODUCTION</b> . . . . .	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Objective . . . . .	5
1.3 A Proposal to Achieve the Objective . . . . .	5
1.4 Text Organization . . . . .	7
<b>2 THEORETICAL BASIS</b> . . . . .	<b>9</b>
2.1 Basic Terminology . . . . .	9
2.2 Static Analysis . . . . .	10
2.2.1 Manual Inspections . . . . .	11
2.2.2 Reverse Engineering . . . . .	11
2.2.3 Static Source Code Analysis . . . . .	12
2.2.4 Automated Static Analysis Tools . . . . .	14
2.3 Software Testing . . . . .	16
2.3.1 Dynamic Analysis . . . . .	18
2.3.2 Black Box Testing . . . . .	19
2.3.2.1 Boundary Values Analysis . . . . .	20
2.3.2.2 Equivalence Partitioning . . . . .	21
2.3.2.3 Random Testing . . . . .	21
2.3.2.4 Pairwise Testing . . . . .	22
2.3.3 White Box Testing . . . . .	23
2.3.3.1 Control Flow Graph . . . . .	24
2.3.4 Model-Based Testing . . . . .	24
2.4 Test Oracles . . . . .	26
2.5 Related Publications . . . . .	27
2.5.1 Static Source Code Analysis and Testing . . . . .	27
2.5.2 Test Oracles . . . . .	29
2.5.3 Considerations of Related Literature . . . . .	31
2.6 Final Remarks . . . . .	32
<b>3 THE REACTOR METHOD</b> . . . . .	<b>33</b>
3.1 Dynamic Information via Test Case Generation . . . . .	34
3.1.1 Automatic Equivalent and Boundary Partition Analysis . . . . .	35

3.1.2	Reduction of Test Cases by Pairwise Testing . . . . .	37
3.2	Static Information via Reverse Engineering . . . . .	42
3.2.1	Regular Expressions . . . . .	44
3.2.2	Source Code Reading . . . . .	47
3.2.3	SUVV Modeling . . . . .	51
3.2.4	Source Code Decomposition . . . . .	52
3.2.5	Source Code Classification . . . . .	55
3.2.6	CFG Generation . . . . .	59
3.3	Source Code Interpreter . . . . .	60
3.4	Static Code Analysis . . . . .	71
3.5	Oracle Procedure . . . . .	84
3.6	Final Remarks . . . . .	87
<b>4</b>	<b>IMPLEMENTATION OF REACTOR . . . . .</b>	<b>89</b>
4.1	REACTOR's Architecture . . . . .	89
4.2	Configuration . . . . .	91
4.3	REACTOR's Execution . . . . .	92
4.3.1	Automatic Test Case Generation . . . . .	92
4.3.2	Automatic Reverse Engineering of Static Source Code . . . . .	96
4.3.3	Automatic Detection of Defects . . . . .	98
4.4	Instrumented SUVV Execution . . . . .	107
4.5	Oracle Procedure Execution . . . . .	107
4.6	Final Remarks . . . . .	109
<b>5</b>	<b>EVALUATION . . . . .</b>	<b>111</b>
5.1	Evaluation Criteria . . . . .	111
5.2	Overall Results . . . . .	112
5.2.1	Case Study 1: Factorial Problem . . . . .	112
5.2.2	Case Study 2: Hanoi Towers . . . . .	112
5.2.3	Case Study 3: Triangle Classification . . . . .	116
5.2.4	Case Study 4: Quick Sort . . . . .	116
5.2.5	Case Study 5: Merge Sort . . . . .	117
5.2.6	Case Study 6: Bubble Sort . . . . .	117
5.2.7	Case Study 7: Insertion Sort . . . . .	117
5.2.8	Case Study 8: Fibonacci Series . . . . .	118
5.2.9	Case Study 9: Arithmetic Mean . . . . .	118
5.2.10	Case Study 10: Threads . . . . .	118
5.2.11	Case Study 11: ORCAS . . . . .	119



5.3	Additional Discussion About the Evaluation	119
5.4	Final Remarks	122
<b>6</b>	<b>CONCLUSION</b>	<b>123</b>
6.1	Requirements and Limitations	124
6.2	Future Work	125
6.3	Final Remarks about this PhD Thesis	125
	<b>REFERENCES</b>	<b>127</b>
	<b>Appendix A</b>	<b>137</b>
A.1	Additional Information About Regular Expressions	137
A.2	Additional Information About Source Code Classification	151
	<b>Appendix B</b>	<b>153</b>
B.1	Additional Information About Evaluation of Hanoi Towers	153
B.2	Additional Information About Evaluation of Triangle Classification	155
B.3	Additional Information About Evaluation of Quick Sort	157
B.4	Additional Information About Evaluation of Merge Sort	161
B.5	Additional Information About Evaluation of Bubble Sort	165
B.6	Additional Information About Evaluation of Insertion Sort	169
B.7	Additional Information About Evaluation of Fibonacci Series	173
B.8	Additional Information About Evaluation of Arithmetic Mean	174
B.9	Additional Information About Evaluation of Threads	178



## 1 INTRODUCTION

The last three decades observed an extraordinary advance in development of computer technology, and this meant that what once was a privilege of few computer enthusiasts and professionals, today it is commonplace and indispensable in our daily lives. Powerful computational resources that were previously used only in extreme cases due to their high costs, can be found today in the pockets of any urban life teenager. Such resources are present ranging from digital music players and smartphones, to complex control systems for trains and huge aircraft. These resources depend pretty much on software, which is a common element that is present in many types of devices used directly or indirectly in everyday's life, and people are surrounded by systems, embedded in devices, that perform some kind of control (CATSOULIS, 2005).

With advances in hardware and software, increasingly complex problems are solved by computer systems in order to increase efficiency towards several aspects: cost, time, productivity, and safety. Computer systems no longer only assist human labor in repetitive tasks, but today they represent complex tasks and even replace humans to take decisions that can involve high risks (SOMMERVILLE, 2010). Consequently, in order to be able to provide safe and consistent decisions operating across several platforms and conditions, modern systems have more and more needs to be verified and validated (PRESSMAN, 2014) in order to become reliable in terms of their functional and non-functional requirements.

Therefore, tests have become an important activity in software development, since needs for quality and robustness have become rather demanding (SOMMERVILLE, 2010). In this context, testing is no longer a development of a subsequent task and has become a very important process with a strong interaction with the project since its beginning (MYERS, 2011).

V&V (Verification and Validation) aims at increasing the quality of software products. Several V&V processes/methods exist, such as static analysis, formal verification, and the most popular method called software testing. So, it is essential even with their costs being usually high (PRESSMAN, 2014). Thus, software testing process may demand a considerable time of any software development lifecycle, but especially in critical applications that, in case of failure, may cause risk to the environment, humans, or high financial costs (SOMMERVILLE, 2010).

Software testing is a high cost process in several cases, and it is still more costly in

space systems that involve a complex and costly hardware engineering, since many of these systems are kept in operation for long periods such as twenty years or more (GRILO et al., 2010).

Research institutes like INPE (*Instituto Nacional de Pesquisas Espaciais*) develop software for embedded computers in stratospheric balloons and satellites. In these applications, the hardware involved often need to cope with problems such as the limited space, weight and energy consumption; so, the development of software embedded in computers of satellites or balloon applications need to take into account such constraints and the critical aspects of such systems require a lot of effort in V&V activities such as static analysis, testing, or formal verification (SANTIAGO JÚNIOR, 2011).

## 1.1 Motivation

The static source code analysis is a safe method to detect defects and has become the focus of several researches in software testing to address deficiencies in specifications. Several studies report that more than 60% of software defects can be detected using source code inspections (SOMMERVILLE, 2010). However, static analysis has two limitations: it is not able to detect defects based on dynamic information, and it potentially can reveal a lot of false positive defects. It is important to stress that if a professional does not agree with certain rule implemented in a static analysis tool not necessarily mean that a false positive indeed exists. However, results by using static analyzers are usually very noisy where there may be lots of false positives and also many defects identified that are not, in fact, important to a particular software product or coding standard. So, static analysis researchers have tried to combine static analysis with some data flow analysis, in order to improve its performance.

Software testing based on a test case generation criteria is probably the most used V&V process in practice. Two known approaches for test case generation are black and white box testing (PRESSMAN, 2014). Black box testing refers to generate test cases based on requirements and without the need to “see” the source code. On the other hand, in white box testing, we usually depend only on the code to generate test cases, by generating representations such as CFG (Control Flow Graph) or Def-Use Graphs (BEYDEDA et al., 2001). In practice, black box testing is the most used in many companies (GAROUSI; ZHI, 2013), and it is also capable to detect defects that can not be addressed only by static analysis or reverse engineering.

Reverse engineering can be described as a form of static code analysis, and the soft-

ware modernization generally uses reverse engineering to understand the actual state of existing or legacy software in order to plan its evolution (LANZA, 2003). Reverse engineering is a process of examination so that one can analyze an SUVV (Software Under Verification and Validation) in order to create another representation at a higher level of abstraction, going backwards through the standard development cycle.

It is not uncommon to find V&V processes integrated concurrently with a reverse engineering. As mentioned before, V&V processes are expensive and essential to develop a high quality software, and at the same time, exhaustive testing is not an option in most of the cases (SANTIAGO et al., 2008a). So, the automation of several activities of the testing process provides benefits, since it can reduce costs both financially and in terms of time required to perform the test process. Therefore, it is essential that efforts are dedicated in order to automate all the activities of the software testing process from test case generation to test results evaluation, which can be assisted by a test oracle (BINDER, 2000).

Therefore, the approach discussed in this thesis is related to static analysis and reverse engineering, which are subjects that currently have great relevance and great challenges to detect types of defects that are not covered only via testing. But it also mixes different black box testing techniques into a single approach that becomes interesting because it is possible to rely on the main benefits of each technique increasing the defect detection capability and enabling the inference of estimated expected results via test oracle generation.

In addition, this approach is proposed by using just the source code as a unique resource, since it is possible to use static analysis techniques (that are generally effective and can be automated as well) to perform its reverse engineering.

The decision to adopt this approach based only on source code is another challenge related to the dependency of software documentation for testing. The quality of tests commonly depend on software documents, e.g. software requirements specification (BINDER, 2000). However, for testing this can be an obstacle in several scenarios where systems are poorly documented, or where documentation is not updated according to the evolution of a legacy system, or even a misspelled documentation which can lead to a misinterpretation of the testing professional. It is really difficult to identify all significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces (SANTIAGO JÚNIOR, 2011), and this needs to be taken into consideration even for well known development

models as waterfall model or V-Model. Considering that tests based on unreliable specifications can not be successful, a more feasible approach would be based on source code that is the more concrete representation of a system. The decision to adopt this approach based only on source code is another challenge related to the dependency of software documentation for testing. The quality of tests commonly depend on software documents, e.g. software requirements specification (BINDER, 2000). However, for testing this can be an obstacle in several scenarios where systems are poorly documented, or where documentation is not updated according to the evolution of a legacy system, or even a misspelled documentation which can lead to a misinterpretation of the testing professional. It is really difficult to identify all significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces (SANTIAGO JÚNIOR, 2011), and this needs to be taken into consideration even for well known development models as waterfall model or V-Model. Considering that tests based on unreliable specifications can not be successful, a more feasible approach would be based on source code that is the more concrete representation of a system.

Finally, the inference of the estimated expected result for an SUVV is not a particular problem assigned in test oracles, but also for other white box testing techniques. For example, one can cite unit testing performed in *JUnit* (JUNIT, 2015), a popular automated test execution framework for testing Java classes. In *JUnit*, a test case script must specify the expected result in order to run and provide an automated verdict. And it must be coded manually by the system tester, as shown in Figure 1.1.

The estimation of expected results is not trivial, since it must consider several possible problems to determine how and whether such an estimate can be produced. The precise inference of results from an SUVV that uses some randomized element is impossible. Scalar variables, usually present in interpreted languages such as Perl and PHP (Hypertext Preprocessor), and non primitive types as arrays, strings, collections or objects (common in object oriented systems) are complex to be inferred. Also, there are SUVV that implement a GUI (Graphical User Interface), and, consequently, might not have predictable inputs or outputs in several situations. And finally, many systems can have as input/output data dependent from database systems or file systems. Therefore, the challenge to develop an approach to infer expected results, by minimizing to some extent manual interference, has currently a great relevance (AGGARWAL et al., 2004).

Figure 1.1 - JUnit test case script.

```
/**
 * Test of factor method, of class Factorial.
 */
@Test
public void testFactor() {
    System.out.println("factor");
    long n = 0L;
    long expResult = 0L;
    long result = Factorial.factor(n);

    assertEquals(expResult, result);

    fail("The test case is a prototype.");
}
```

## 1.2 Objective

The objective of this PhD thesis is to present a solution to detect a wider range of software defects via the combination of static source code analysis and testing. Such a combined solution may have the strengths of both methods, static code analysis and software testing, and the increased ability to detect a greater range of software defects is interesting not only to ordinary but also to critical software systems.

## 1.3 A Proposal to Achieve the Objective

In order to reach the objective of this work, a combined approach that uses static source code analysis and testing was developed. The created method and tool is called REACTOR (Reverse Engineering for stAtic Code analysis and Testing to detect sOftwaRe defects) which uses reverse code engineering to support both the methods (static code analysis and testing).

For static source code analysis, the main contribution of this PhD thesis is the use of dynamic information in order to complement the usual way to perform static analysis. For testing, REACTOR is related to two testing process activities: test case generation and test results evaluation (the oracle problem). With respect to test case generation, the benefits of three black box testing techniques - BVA (Boundary-Value Analysis), EP (Equivalence Partitioning), RT (Random Testing) - are an adapted version of the original combined into a single approach. In addition, pairwise testing (LEI; TAI, 1998) is used in order to decrease the size of the test suite. For test oracle, a

solution based only on the source is presented where there is no delivery of a precise oracle information, but the test case generation and test oracle activities together provide a solution to address, more specifically, exception handling defects that turn into failures.

By combining static code analysis with software testing, REACTOR has a potential to detect a wider range of defects. The proposal was applied to 11 case studies, including one from the space application domain. With respect to static code analysis, REACTOR performed better when compared with three other well known static code analyzers considering the detected true positives and unimportant defects. For testing, REACTOR detected exception handling defects considering the 11 case studies.

Case studies used in this work were developed in the Java programming language. Java and its derived languages still seem to be most used programming languages in the real settings ([REDMONK, 2016](#); [TIOBE, 2016](#)). Though, C is the most used programming language for embedded systems ([UBM TECH ELECTRONICS, 2014](#)).

Java can be the solution for the development of simulators of systems as part of an EGSE (Electrical Ground Support Equipment), including space applications. But, it is important to point out that most of the concepts and the scientific contribution of REACTOR can be adapted to source code developed in other programming languages that support the object-oriented programming paradigm such as C++, PHP, and C#.

The theoretical contributions of this work are:

- a) Development of a method combining two V&V approaches: static code analysis and software testing;
- b) Handling several types of static code analysis defects and testing defects as bad coding practices, code parts that causes a performance losses, unused code parts, coding vulnerabilities, and code lines that can cause failures.
- c) Development of a new test case generation method based on BVA, EP, and RT;
- d) Development of an organic test oracle that does not depend on explicit documentation.



Apart from the theoretical contributions, the implementation of this approach into a tool, REACTOR, brought several practical contributions:

- a) The application of reverse engineering in order to perform automated source code inspection;
- b) Using regular expressions patterns to match code lines of the SUVV;
- c) The estimation of expected results (oracle information) automatically without the need of explicit documentation;
- d) Generation of test scripts by automatically instrumenting the source code;
- e) Application of the approach to 11 case studies including a simulator of software embedded into satellite scientific instrument's computer;

#### 1.4 Text Organization

This work is divided into chapters whose division is described below:

- **Chapter 2 - Theoretical Basis:** briefly describes several issues related to basic terminology, static analysis, software testing, testing techniques, reverse engineering, test automation, and test oracles;
- **Chapter 3 - REACTOR Concepts:** discusses in details the concepts behind the REACTOR method: static source code analysis, software testing and reverse engineering;
- **Chapter 4 - Implementation of REACTOR:** discusses the features and other implementation level characteristics of the REACTOR tool, the realization of the concepts presented in the previous chapter;
- **Chapter 5 - Case Studies:** presents results obtained by REACTOR considering a set of case studies;
- **Chapter 6 - Conclusion:** presents conclusions of this PhD thesis and possible future directions for this research.

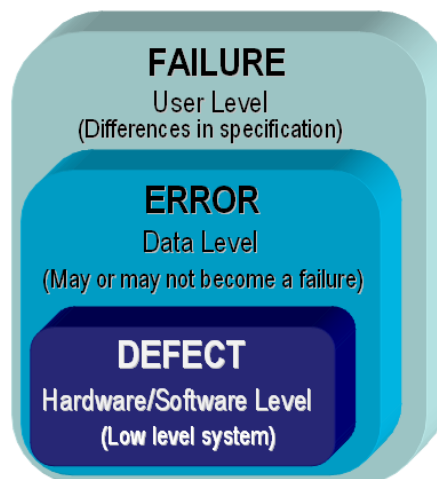


## 2 THEORETICAL BASIS

### 2.1 Basic Terminology

In order to avoid possible confusion in understanding the terminology in the area of software testing, this sub-section provides some basic definitions. In fact several authors in software engineering area distinguish these terms (defect, fault, error and failure) in different settings, as there is no consensus among all authors. In this work, defect is considered a synonym of fault. This terminology is defined in accordance with the *IEEE Standard Glossary of Software Engineering Terminology* (INSTITUTE..., 1990) and can be understood by Figure 2.1.

Figure 2.1 - Defect, Error and Failure Terminology.



The defect (or fault) is the occurrence of an incorrect step, process, or data definition in a software. It occurs in a lower level of application, a defective memory that changes a bit, corrupted file, or a syntax problem from the compiled program. An error is the effect of lack of data manipulated by the program, and this error may or may not manifest a failure, that is the user perception of the occurrence of a defect (BINDER, 2000).

Processes of V&V ensure that the software satisfies all the requirements necessary to develop all tasks for which it was designed (Validation), and to ensure that all requirements have been met (Verification) (PRESSMAN, 2014). Briefly, one can say that there are inspections with intention of certifying that the system satisfies user

expectations. One of the most classical definitions of V&V was written by [Boehm \(1981\)](#) and says that validation is the answer to the question “are we building the right product?”, and the verification replies to “are we building the product right?”.

Some other definitions used in this PhD thesis are defined according to the glossary of IEEE ([INSTITUTE... , 1990](#)) and ([MYERS, 2011](#)).

- **Input:** Specific data value to be entered during the execution of a particular test case.
- **Output:** Data resulting from an operation, software execution, or test execution.
- **Test Case:** Set of finite entries assigned to perform a system test.
- **Test Suite:** Collection of test cases that are intended to be used to test an SUVV.
- **False Positive:** It is a false alarm that a software defect was found when in fact it does not.
- **Return Value:** If a statement (or method on object oriented systems) contains an output variable, this is treated as a return value.
- **Expected Result:** Observable conditions or states expected as a result of running a test.
- **Driver:** Software, or any other mechanism, which is used to apply test cases on SUVV.

## 2.2 Static Analysis

Static code analysis is the process of detecting defects in a software’s source code. The term “static analysis” can be viewed as an automated code review process for evaluation of source code, or other system representation, without executing it ([CHESS; WEST, 2007](#)). Similar to manual inspections, automatic static analysis is a technique where the source code of an SUVV is checked searching for particular patterns that are automatically classified as potentially defective ([SOMMERVILLE, 2010](#)).

### 2.2.1 Manual Inspections

It is interesting to mention that the term “inspection” (or code reviewing) is generally related to the analysis performed by a human, because when the analysis is performed by a software tool, the most common term is “static analysis”. Inspection and peer review are the most commonly used static analysis techniques, where group of people check specification, design, or source code. They examine the design or source code in detail, looking for possible defects or omissions.

The inspection works well assuming that programmers can notice defects in somebody else’s source code much easier than in their own source code. However, although it is a very efficient technique for detecting defects, the dependence on manual inspections has several issues:

- Manual work usually produces inconsistent and fragmentary test process information. Thus, the testing team may produce, for instance, inconsistent system test cases due to unambiguous requirements not detected during the manual inspection;
- It is difficult to accomplish an appropriate manual traceability between requirement specifications and other artifacts derived during the development of the software product;
- It is an extremely expensive, since it requires several inspectors at regular times;
- There is a dangerous risk that a test success depends on an expert professional exclusively;
- In most cases, manual inspection is tedious and error-prone;

Manual inspections are an effective technique to detect defects, but in most of the cases, it is extremely expensive and some times infeasible. However, it is possible to use automatic and more efficient approaches to aid testing inspections by the use of reverse engineering to create architectural views of SUVV.

### 2.2.2 Reverse Engineering

Reverse engineering is the processes of extracting knowledge from anything already made in order to reproduce anything based on this extracted knowledge.

In practice, there are two main types of reverse engineering. The first type is where there is no source code available, and the engineering is focused on several efforts towards discovering the possible source code for the regarded SUVV. This usage is more familiar to most of the people. And the second type, which is implemented in this work, that is when system's source code is already available, but higher-level aspects of the SUVV may be discovered due to poor or outdated documentation.

For a software to be completely understood, it is necessary to extract two types of information: static and dynamic (GRILO et al., 2010). Static information are closely related to white box approach, since the information can be basically recovered by source code analysis, such as software elements (classes, methods and variables) and relationships between them. Relationships can be complex spanning the extension between classes, interfaces, and overwritten or overloaded method calls. The dynamic information is related to black box approach. It goes beyond static software elements and can represent the sequence, concurrence and coverage of encoding (GRILO et al., 2010). Thus, reverse engineering can be made focused on two approaches:

- *Static approach*: refers to system verification techniques that do not involve executing the program (MOORE, 1996) and requires access to system's source code. This approach is especially useful for extracting information about the internal structure of the system and dependencies between its elements, so it is classified as a white box approach.
- *Dynamic approach*: is also based on an analysis of system's external behavior, and it is done during execution (MORI et al., 2002). It is the only feasible approach when source code is not accessible, and adequate for the extraction of system's physical structure and its dynamic behavior. This approach is commonly used in black box tests.

### 2.2.3 Static Source Code Analysis

Static source code analysis, also commonly known as code reviewing, is one of the classic, oldest and safest methods to detect defects and it recommends on how to improve the code (CHESS; WEST, 2007). This process can reveal defects and source code fragments that may turn into a failure in the future.

The automated static analysis tools can assist programmers and aims to solve the two major disadvantages of the inspections: its high costs to analyze a code by hiring programmers, and the analysis repeatability. After all, people need to rest regularly,

as their attention uses to weaken quickly when they review a lot of code at a time, thus compromising the quality of analysis. So, static analysis tools are strongly recommended since they can tirelessly analyze large source code routines and give recommendations on which code fragments the programmer should consider.

In many cases, a static analysis tool can not replace a well done code analysis performed by a team of professionals, but in general, the trade off cost/benefit makes the usage of such tools a good practice exploited by many companies. In general, static analysis is related to four problems: detection of defects, recommendations on code formatting, software metrics and reverse engineering.

The SSCARE (Static Source Code Analysis by Reverse Engineering) is possible once a lot of static information can be obtained by static analysis, such as: methods called by other method (or the same method), uninitialized variables, variables set but not used, source code segments that are isolated and not executed by any test case, questionable or unsafe coding practices, among others (CHESS; WEST, 2007). Static approaches are particularly well suited for extracting information about the internal structure of the system and dependencies among structural elements from the source code by reverse engineering.

Based on source code, SSCARE can detect defects that are often a result of programming mistakes or omissions, so they highlight anomalies that could go wrong when the program is executed generating an error or failure. However, many times these anomalies do not necessarily result in an error or failure with test cases, or even in any case, as explained in Section 2.1. This is typically conservative technique, since it reports not only defects that are guaranteed true, but also weaker defects than can (or can not) be true (ERNST, 2003). So, the usefulness of this technique is sometimes questionable, due to the large number of false positives that can be found, especially when analyzing large systems. This is certainly one of the greatest issues of static analysis (CHESS; WEST, 2007).

Other very important issue is that automated tools analyze the source code without considering the possibilities that can be explored through all the computations involved. So, the use of automated tools still has a limited range since some classes of defects can only be identified by making inferences about the control flow data, and computing all possible values for the data (SOMMERVILLE, 2010). For example, which code block within a control structure is actually exercised may depend on the data that the SUVV is handling. This information can only be addressed by dynamic analysis, which consists in monitoring variable values and instrumenting the source

code to produce information regarding exercised paths (AGGARWAL; JALOTE, 2006).

As static analysis is criticized for revealing false alarms many times, on the other hand, dynamic testing operates by executing (or simulating an execution) a program and observing its behavior (CHESS; WEST, 2007). So, this is precise because it examines the exact run-time data, and there is little or no uncertainty in what control flow paths were taken or what values were computed once selected a representative set of test cases.

Traditionally static and dynamic approaches have been viewed as separate domains, with practitioners or researchers specializing in one or other. However, the difference is smaller than it appears, and it is certain that these distinctions are unnecessary and counterproductive. Hence, several researches aims to use static and dynamic approaches as complementary techniques, and showing that there is a synergy between their strengths and weaknesses (ERNST, 2003). However, in terms of existing automated tools, they are still limited to perform only the source code analysis based only on static information.

#### **2.2.4 Automated Static Analysis Tools**

In order to be effective and repeatable, testing must be automated. And tests can be automated by some kind of software that includes capabilities to generate test inputs and to run test suites without manual intervention of a software tester (BINDER, 2000).

The more appropriate approach of automated testing depends directly on the goals, budget, software process, class of application under development, and particularly limitations of the development and target environment (BINDER, 2000). An useful test automation can be entirely different for embedded systems, graphical interfaces or database based systems, for example.

Automation of testing must be timely used as much as possible in an effective manner, since it offers many significant advantages:

- It allows a faster and efficient detection of defects significantly reducing the amount of post-release updates;
- Costs of test automation are generally recovered increasing productivity and avoiding costs associated with defects correction;
- Reduces the risk that test success depends exclusively on an expert;



- Tester productivity is improved with an increase of time for designing tests achieving a greater coverage;
- Automated evaluation is the only efficient manner to evaluate a huge amount of outputs by repetition.

For several types of defects, it is possible to automate the process of checking source code revealing code fragments that may be defective. These techniques were employed in development of automated static analysis tools, which allows testers to perform an automated analysis without a complete knowledge of the SUVV and without the need for much information. Such tools complement the defect detection facilities being used as part of the inspection process or as a separate verification process activity, since they are faster and cheaper than detailed code reviews. Some examples of well known static analysis tools are: *SciTools Understand*, *SonarQube* and *FindBugs*.

*SciTools Understand* (SCITOOLS, 2015) is a very advanced commercial tool IDE built focusing the code knowledge. It does not only perform static analysis tests but also has a lot of features related to metrics, editor, graphing, dependency analysis, among others. This tool is able to perform tests in several programming languages as COBOL, C, C++, Fortran, Java, Pascal, Python and PHP. It depends only on the source code, but the developer does not provide a list of all types of defects that the tool can verify.

*SonarQube* (SONARQUBE, 2015) is an open source platform which provides code analyzers, reporting tools, defects hunting modules, among others. In terms of programming languages, Sonar supports C, C++, C#, COBOL, Groovy, Java, PHP, Python, VB (Visual Basic) and many others. Different from *Understand*, *SonarQube* is less dependent on code knowledge and more on static analysis, and the only requirement to perform tests in it is the source code. This tool runs hosted in a server and it has a web-based GUI. So, its installation and setup are not as trivial as the previous.

*FindBugs* (FINDBUGS, 2015) is an open source static code analyzer which detects possible bugs in Java programs. This tool is distributed as a stand-alone GUI application, and rather than other two tools, *FindBugs* operates on Java bytecode instead of the source code. As can be seen in Section 2.5.1, this is probably the most referenced static analysis tool in academics, since it is easy to install, easy to use, and very well documented. It also has plug-ins available for some IDEs, and its only

limitation is that it performs static analysis of Java code only.

### 2.3 Software Testing

Testing is a set of processes intended to ensure that a software does what it is intended for, and to discover possible defects before it is put into use (SOMMERVILLE, 2010). Therefore, testing process has two main objectives: to demonstrate to the developer and the customer that the software meets its requirements, and to reveal situations in which the behavior of the software is incorrect, undesirable, or does not match to its specification (SOMMERVILLE, 2010). The occurrence of defects can invoke an undesirable software behavior, software crashes, unwanted interactions, incorrect computations, and data corruption (SOMMERVILLE, 2010).

The first objective leads to validation (V&V testing), where the software is tested by using a given set of test cases which may reflect the expected software behavior (SOMMERVILLE, 2010). And the second objective leads directly to detection of defects, where test cases are designed, on purpose, to expose possible defects. So, the test cases for detection of defects can be deliberately nonsense and do not need to reflect exactly how the software may be used (SOMMERVILLE, 2010).

However, in real situations, there is no exact boundary between these two objectives. Thus, it is not uncommon that defects are revealed during validation testing, and otherwise, it is not unusual to discover situations where software does not meet its requirements during tests for detection of defects (SOMMERVILLE, 2010). And this is the reason that is necessary to understand the validation (V&V) even in research dedicated to the detection of defects, as this thesis.

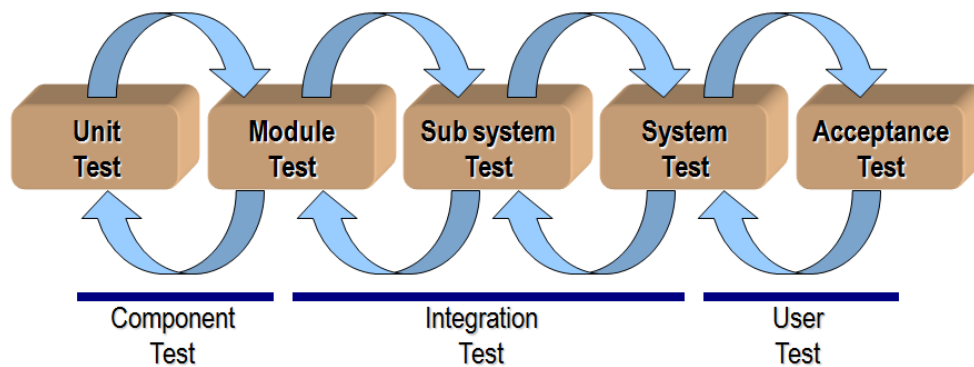
V&V is a technical and systematic evaluation at the end of each development stage, to ensure that all requirements have been complied with. Design, source code, documentation and data must satisfy these requirements undergoing revisions, walk-throughs and testing. One can say that the focus of verification is in the process, while the validation is in the product (SOMMERVILLE, 2010).

Such tasks are usually more effective when conducted by people not directly involved in the product development, since testers involved in the project can unconsciously develop biased and well behaved tests that do not exploit system limits, and often do not represent the environment for real operation. Walkthrough is a closer examination based on debugging source code with the intent of finding errors. It is common that V&V teams have, besides test developers, experts in quality assur-

ance. Tests are conducted operating the SUVV with real data inputs, produced by developers, and running SUVV in a real situation. This simulation demonstrates that the software satisfies all requirements, and if not, identify differences between results obtained compared with expected results.

V&V process is usually composed of five stages which are tested from smaller components up to the full integrated system. This process is interactive, and based on feedbacks, it is possible to return to an earlier process stage in order to fix defects in its sources satisfying system requirements. These five stages are illustrated in Figure 2.2.

Figure 2.2 - Test Process.



SOURCE: Adapted from Sommerville (2010).

- **Unit Testing:** Tests of individual components independently. In an object-oriented methodology, a unit could be a method or a class;
- **Module Test:** Tests of collection of dependent components, such as a class hierarchy;
- **Subsystem Test:** Test of a set of integrated or subsystem modules, or sub-packages in an object-oriented methodology;
- **System Test:** Testing of integrated subsystems that compose a system, or a package;
- **Acceptance Testing:** System testing with real data instead of simulated data.

Therefore, tests for V&V can be applied at different stages of the software development process, and there are several documented techniques for planning and validation of test cases (MYERS, 2011). Each of these techniques has a different perspective of the SUVV, and together they complement each other (PRESSMAN, 2014).

### 2.3.1 Dynamic Analysis

Ball (1999) defines dynamic analysis (or dynamic testing) as “the analysis of the properties of a running software system”. In other words, dynamic analysis is the study of a software execution, and has become a common technique which has received a lot of attention from the software testing community. The proper analysis of data gathered by executing software has potential to provide an accurate picture of an SUVV, since it exposes its real behavior. This picture can expose runtime information and object identities (in context of object-oriented systems) for scenarios that are exercised during the analysis (CORNELISSEN et al., 2009). Dynamic approaches are the only option when the source code is not available, and they are usually more difficult to automate.

The SUVV comprehension is another purpose of dynamic analysis, and several approaches have been proposed in this context using different interpretation techniques and tools (CORNELISSEN et al., 2009); and the source code instrumentation is a useful resource to produce information regarding exercised paths in order to detect software defects (AGGARWAL; JALOTE, 2006).

Generally, dynamic analysis also comprises the analysis of an SUVV execution through interpretation or instrumentation, and the resulting data are used for such purposes as reverse engineering and debugging. It does not only include software artifacts, but also contains other essential information to comprehend SUVV as sequential information, information about concurrency, code coverage, etc.

The dynamic analysis operates out of the limited range of static analysis, since it involves executing test cases and evaluating the results by monitoring variable values (CORNELISSEN et al., 2009). For example, which code block within a control structure is actually exercised depends on the data that the SUVV is handling, and this information can only be addressed by dynamic analysis.

Therefore, dynamic analysis can reduce the search space for static analysis by restricting it only for the source code exercised by test cases, saving the tester time that is not interested in analyze useless code. Thus, the hybrid approach tends to

present a better reliability of the results, since dynamic analysis can be used to confirm or discard results from static analysis (WENDEHALS, 2003). In short, in this work dynamic analysis help to make results of static analysis more precise.

### 2.3.2 Black Box Testing

Also called “functional testing”, is based only on the software specification or performing tests without knowledge of its internal structure; therefore, the tester has no access to the source code. The purpose strictly ensures that the system is able to perform all functions required no matter how they implemented their solutions. The main types of tests used in this approach are:

- **Boundary Value Analysis:** Consists of selecting data (or cases) for tests beyond software’s range of values. The extrapolation of maximum and minimum values are often the most responsible for failure occurrence.
- **Equivalence Partitioning:** Divides the set of input data into classes that are tested from specific cases. The main objective of this technique is to eliminate redundancy of test cases optimizing the discovery of defects with less effort from testers.
- **Category-Partition:** Consists of a systematic way to design functional test cases. Significant test cases are generated based on combining values, input parameters or operating environment which are inferred by analyzing the system specification.
- **Classification Trees:** It is based on the partition of the input domain of a test object, which is considered under various aspects addressed by the tester if it is relevant. So, test cases are composed by combining classes of different classifications.
- **Random Testing:** As the name suggests, this technique tests the SUVV by using the generation of random and independent inputs.
- **Cause Effect Graphing Technique:** Graphs, in general, are important tools for software testing. This particular technique uses graphs to construct a representation (or model) of logical conditions and their corresponding actions.

It is important to point out that very classical black box test case generation techniques, e.g. BVA, are still the most used in practice in many companies (GAROUSI;

ZHI, 2013). Mixing different black box testing techniques into a single approach can be interesting because one can rely on the main benefits of each technique.

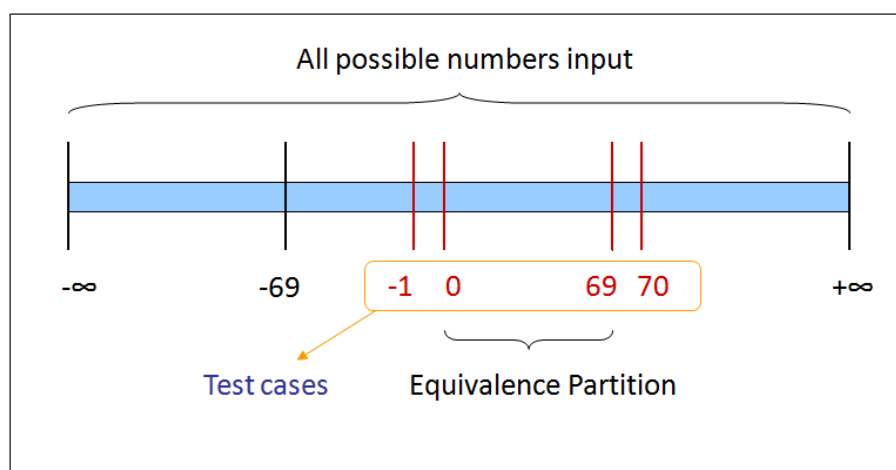
### 2.3.2.1 Boundary Values Analysis

BVA is an usual testing technique for functional tests where each boundary condition of an equivalence class is tested (MYERS, 2011). Several experiences show that test cases that explore boundary conditions of an SUVV have a greater efficacy than test cases that do not (MYERS, 2011).

In this case, boundary conditions are those situations directly within, above, and below the limits of an equivalence class. BVA is different from EP, since rather than selecting any value in an equivalence partition, BVA requires that one or more values be selected as test case in order to test each threshold of the equivalence class.

Figure 2.3 presents an example of the BVA set to test the factorial in a handheld calculator. It is known that most of handheld calculators can just calculate factorial numbers up to 69 at most, due to memory limitations. And also, it is known that there is no factorial for negative numbers. So, based on BVA testing, the factorial for this calculator could be reasonably tested with four test cases. Two of them are the edges within the equivalence class, and other two immediately outside the boundaries.

Figure 2.3 - Example of an BVA application.



### 2.3.2.2 Equivalence Partitioning

EP is a testing technique that aims to reduce the number of tests by setting classes of values. Therefore, this technique assumes that a test of an SUVV with a representative value within a determined class, consequently tests all values within that class (MYERS, 2011). EP assumes that a well planned test case has a reasonable probability of finding defects avoiding exhaustive tests, that is impossible in most times.

When testing an SUVV, generally one is limited to a small subset of possible inputs. Hence, it is needed to select the “right” subset, which is the one with the highest probability of revealing the most number of defects. One way of choosing this subset is by considering that a well planned test case may have two characteristics:

- It may drastically reduce the number of test cases that must be developed to meet some predefined requirement for a reasonable testing.
- It should cover a large range of other possible test cases, allowing one to foresee (of course, without the absolute certainty) the defects that may occur with input values within the same equivalence class.

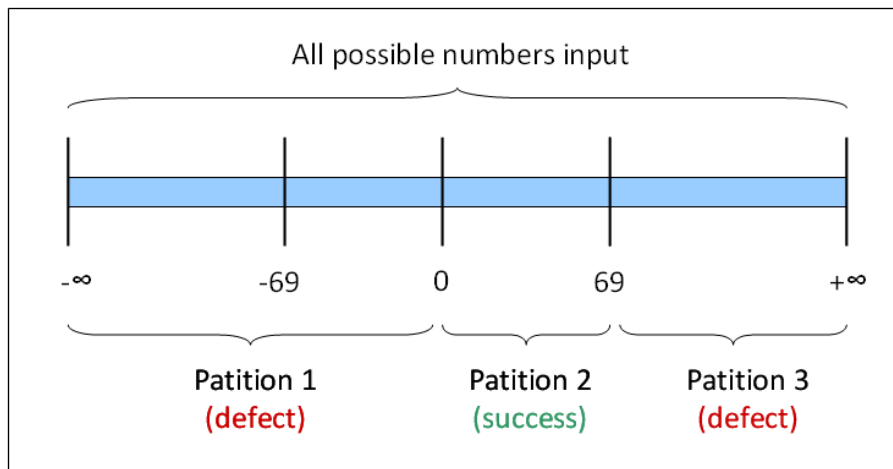
EP suggests that tests with a representative value within a determined class is equivalent to a test with any other value within the same class. So, if one test case within an equivalence class detects a defect, it is expected that all other test cases within the same equivalence class would reveal the same defect. And logically, if a determined test case did not detect any defect, it is expected that no other test cases within the same equivalence class do not also.

Figure 2.4 presents the same example of the handheld calculator, that is in mentioned in Section 2.3.2.1, treated by EP approach. In this approach, the factorial could be reasonably tested with three test cases, one for each partition set in Figure 2.4.

### 2.3.2.3 Random Testing

Random testing (also known pejoratively as “monkey testing”) is a functional (black box) software testing technique where the SUVV is tested with random and independent inputs generated somehow (GODEFROID et al., 2005). This technique is specially useful in situations where the time needed to design test cases is too long, or the complexity of test cases makes it impossible to test every possible combination of inputs.

Figure 2.4 - Example of an EP application.



Although RT is usually considered a inefficient approach of software testing when compared with test case generation based on software structure (which is preferred in most of times (DURAN; NTAFOSS, 1981)), there are several researches that presents experiments based on random testing which tend to confirm the viability this testing technique as a useful validation tool (GUTJAHR, 1999).

#### 2.3.2.4 Pairwise Testing

Pairwise testing (LEI; TAI, 1998), also known as 2-way testing, is a known test case generation criterion that requires that for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case. The generation of the minimum pairwise test set is considered a NP-complete problem.

Empirical results show that pairwise testing is a practical and effective approach that tends to use the least amount of variable combinations keeping an effective coverage of the tests. It groups the variables in pairs for test suites, assuming that the interaction of at most two values can cause the most of defects. A practical example of using pairwise testing demonstrating its capacity to reduce test cases is shown in Section 3.1.2.



### 2.3.3 White Box Testing

Also known as “structural testing”, is performed at a lower level than functional tests, and is based on internal structure of the system by accessing source code implementation (SOMMERVILLE, 2010). The tester needs full access to system’s source code in order to examine logical paths and verify its operation. At this level, there is no concern about functional requirements or system specification (PRESSMAN, 2014). White box tests can detect many implementation failures or even highlight never executed paths (useless code). The main types of tests used in this approach are:

- **Control Flow Graph:** Consists of directed graphs which has nodes that are blocks of sequential statements. Edges contain a label or predicate that represents the condition of control transfer. There are other several techniques based on CFG, as Concurrent CFG, Hierarchical CFG, among others.
- **Def-Use Graph:** Similar to a CFG, it captures the flow of definitions (computation-use or c-use and predicate-use or p-use) to each node in the graph and uses across basic blocks in a program. It also labels each edge with a condition that causes the edge to be taken if it is true.
- **Basis Path Testing:** Aims to define a basic set of testing finite paths. Requires the creation of a set of test cases that exercise the paths guaranteeing the execution of each statement contained in these at least once.
- **Condition Test:** Tests logical conditions (*if, for, while, etc.*) contained in the code. Conditions can be simple (with only one logical operator) or composed.
- **Data Flow Testing:** Makes execution paths selection according to definitions and locations of variables or, in other words, data flow is selected based on the data manipulated in certain code snippets.
- **Bounds Test:** The tester makes use of test cases that traverse program’s bounds, in order to validate the construction of each bound.

There is another technique called “Error-based testing” which is based on the introduction of common mistakes in software during the development process. Many

references consider this technique as belonging to the same group of structural tests (or white box tests).

Structural tests are done generally during the early stages of development in small software parts (functions, methods, or classes); functional tests are done during the integration of software stage checking the outputs for generated test cases at an earlier stage.

### 2.3.3.1 Control Flow Graph

CFG is a known computing technique where a representation of the software is decomposed into a set of blocks such that the execution of a block command can perform the execution of other block commands (MARINKE, 2012). CFG establishes a relationship between nodes and the graph blocks, where each node and edge respectively of the graph represent the command and possible path of the software. So, from the graph, it can choose which components should be run (MARINKE, 2012). Figure 2.5 shows the representation of nodes and arcs generated to the following standard programming structures: *if*, *while/for*, *repeat-until* and *switch-case*, respectively.

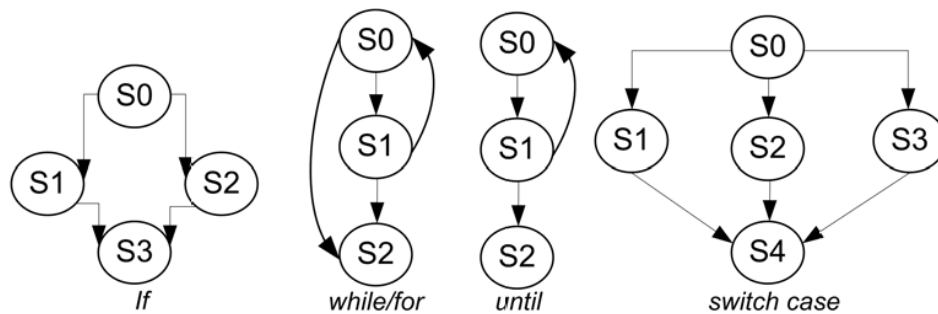


Figure 2.5 - Usual control structures illustrated as CFG.

SOURCE: Adapted from Marinke (2012).

### 2.3.4 Model-Based Testing

Specifying software requirements, complex or not, is commonly designed making use of NL (Natural Language) and can be divided into three types (SOMMERVILLE, 2010).

- **Functional Requirements:** Function statements whose system should provide how it react to its inputs, or in some cases, even statements of what the system should not do.
- **Non-functional Requirements:** Restrictions on services or functions provided by system as patterns, development constraints, response time to an input, etc.
- **Domain Requirements:** In most cases, same as functional requirements, however specific for application domain commonly following standards or restrictions.

Specifications are usually written in NL and often are interpreted for generation of models. Since they are based on a set of consistent rules, graphical or textual language can be used to represent structured information composing a formal model. Models have become common for specification of specific domain applications, particularly in software development (XIAO et al., 2007). However, the process to formalize a specification in a model is not a trivial task and requires time and experience from testers. The SOLIMVA methodology (SANTIAGO JÚNIOR, 2011), for example, was presented to be used in order to minimize the effort spent by the tester in understanding formal system specification. In short, if there is the availability of a reliable specification of requirements, one can use a *top-down* approach and obtain a formal model on the lower level through the interpretation of the requirements for the tester. On the other hand, one can also use a *bottom-up* approach to generate requirements models at a higher level based on some reading or analyzing the source code.

MBT (Model-Based Testing) is a much discussed topic by software engineering authors, because depending on the type of application, the use of a given representation model becomes more similar to a real system. One can cite models known as the ERD (Entity-Relationship Diagram) (CHEN, 1976) that has as main objective the organization and links between data, the DFD (Data Flow Diagram) (GANE; SARSON, 1979) that deals with the relationship among processes performed by system, and Class Diagram (UML, 2015) that describes the structure of a system by showing its classes, attributes, methods (or operations), and relationships among these objects.

One must also mention FSM (Finite State Machine) based models, that represent states of a system and events (or stimuli) that cause transitions from one state to

another, being particularly useful to demonstrate stimuli driven systems. Ward and Mellor (1985) proposed the use of modeling states for designing real time systems, and several other authors quote techniques based on FSM such as Petri Nets (PETERSON, 1981), SDL (Specification and Description Language) (ELLSBERGER et al., 1997) and Statecharts (HAREL, 1987). The major problem with this approach is that the number of possible states for a representation can increase rapidly depending on the complexity of the modeled system.

These techniques are extremely useful, almost any application that makes use of database has an ERD diagram, for example. The choice of which to use depends on the aspect wanted to approach the SUVV, since none of these techniques can alone satisfy all possible aspects to build a complete suite of tests.

## 2.4 Test Oracles

Dictionaries define the word “oracle” as a person considered to be a source of wise counsel or prophetic opinions, or an authoritative or wise statement or prediction. Simply, an oracle has answers to all the questions.

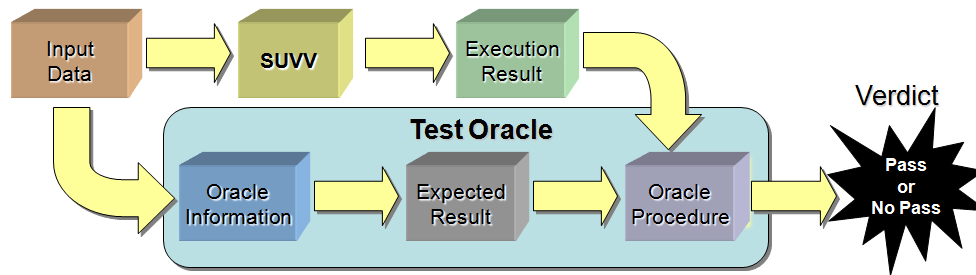
In software testing, oracle is a tool that has been widely used as a framework of expected results for certain test cases applied to the system under test (BINDER, 2000). An oracle is like a set of tuples where for each possible input, an expected output is associated. Thus, one can compare the output obtained from a system execution with the expected output, facilitating the job of testers in detecting faults.

Test oracles are mainly composed of two components, *oracle information* and *oracle procedure* (XIE; MEMON, 2007). *Oracle information* is the information source of expected results, while *oracle procedure* is responsible for comparison.

*Oracle information* is the most complex component of an oracle, since different approaches can be used in its development. Its structure depends directly on the application’s complexity and the range of inputs domain, and these information can be acquired from human knowledge or even via software. In its most common forms, oracles tend to be based on the system’s specifications, tables of examples, or even in knowledge of developers who may know how the software should work for certain cases (HOWDEN; EICHHORST, 1978). The comparison mechanism (*oracle procedure*) is responsible for the verification of results and determining the verdict (XIE; MEMON, 2007). Figure 2.6 illustrates this definition.

Based on several approaches cited by Binder (2000), it can be deduced that there

Figure 2.6 - The basic structure of a test oracle.



is no general rule or “silver bullet” for the oracle development. And due to the complexity that a TOG (Test Oracle Generator) development can offer, a reflection regarding its benefits in face of the implementation effort is perfectly justifiable.

In the case of this thesis, none of the approaches mentioned by Binder (2000) can be totally implemented due these two requirements: it must be based only on source code, and it must work automatically. However, it is possible to classify REACTOR in the group of *Organic Oracles*, since it has as a main characteristic the fact that it does not need an explicit indication of the expected result. *Organic Oracles* exploit information about SUVV structure and the knowledge about the test cases, assuming that it is sufficient to decide whether the result is right or not, exactly as REACTOR does.

Although an extensive automation of the testing process is, in many cases, very dependent on the expected result (oracle information), and the capability to perform an automatic comparison between the expected and actual results (BINDER, 2000), there are approaches that do not depend on expected results. This is the case of organic oracles where the test input data and the actual results are enough to decide the verdict of the test case. This implies that the expected result does not need to be explicitly provided in this way.

## 2.5 Related Publications

### 2.5.1 Static Source Code Analysis and Testing

In this section, it is presented some studies related to static source code analysis and testing (black box, white box). Since these research fields have been extensively addressed, we mention some approaches more related to our work.

Aggarwal and Jalote (2006) proposed a hybrid approach that combines static and dynamic analysis like the one used in REACTOR, but it used a combination of two independent tools for implementations in C, which are not user-friendly.

Balzarotti et al. (2008) also used the same hybrid approach to detect defects in web systems written in PHP, in order to solve SQL (Structured Query Language) injection and cross-site scripting. A tool named Saner was implemented and it found defects in known web applications as *Jetbox*, *MyEasyMarket*, *PBLGuestbook*, *PHP-Fusion 6.01*, *Sendcard 3.4.1*.

Another similar hybrid research was released by Jovanovic et al. (2006), demonstrating Pixy tool, which can identify vulnerabilities in web applications written in PHP. Several open source web applications in PHP as *DCP Portal 6.1.1*, *MyBloggie 2.1.3beta 3* and *TxtForum 1.0.4-dev* were tested in order to demonstrate the efficiency of the tool.

Chatzieftheriou and Katsaros (2011) compared some tools that perform static analysis of source code in C. They are: *Splint*, *UNO*, *cppcheck*, *Frama-C*, *C++ Test* and *Com. B*. These six tools were compared based on the detection of capabilities of certain types of defects such as memory used, and time spent for analysis. The conclusion is that open-source tools generally do not have the same effectiveness of commercial tools.

Another interesting comparison was done in Emanuelsson and Nilsson (2008) to evaluate *PolySpace Verifier*, *Coverity Prevent* and *Klocwork K7M*. These tools were applied to detect defects in implementations in C and C++, and they were analyzed in depth, particularly with respect to their limitations.

Several investigations are focused on detecting some specific defect classes. Li and Cui (2010), focused the work on detecting bugs generated by copy-pasted blocks by programmers in source code by developing *CP-Miner* was used to detect failures caused by copy-pasted code.

The detection of defects in critical systems has also a significant literature. Torri et al. (2010) focused on embedded systems, and did an extensive research on free/open source static analysis tools. The conclusion is that, although there was a certain gain in defect detection, the open-source tools are not sufficient for testing complex embedded systems yet.

Among the free/open source static analysis tools, *FindBugs* is presented in several

academic articles. [Ayewah et al. \(2007\)](#) and [Schmeelk \(2010\)](#) obtained good testing results with it. And [Shen et al. \(2011\)](#) developed the *EFindBugs*, an improved version of *FindBugs* adding a ranking of defects that aids the classification of defects for the report.

[Lucca et al. \(2002\)](#) proposed a black box testing approach for web applications by using decision tables as a combinatorial model for representing the behavior of the Web application and which is applied to generate the functional test cases.

[Arcuri et al. \(2010\)](#) proposed the use of a black box approach by automating tests of real-time embedded systems based on environment models. These models describe some of the structural and behavioral properties of the environment which interact with SUVV, and test cases can be automatically selected based on the models, using various heuristics that maximize chances of fault detection.

[Lapierre et al. \(1999\)](#) investigated several strategies towards to a practical test data generation, and they presented an approach based on the white box approach and symbolic execution. So, execution trees are symbolically executed to produce paths, which can be mapped by an algorithm whose solutions corresponds to the test data used (as input) to cover program branches.

The use of white box testing is also common in web applications. [Liu et al. \(2000\)](#) proposed a white box technique that exploits a data-flow models focused on tests of web applications implemented in HTML (HyperText Markup Language), XML (eXtensible Markup Language), and other scripts. [Tonella and Ricca \(2004\)](#) presented a white box solution based on a control flow model which represents the internal structure of web pages in terms of the execution order.

### 2.5.2 Test Oracles

The following investigations describe proposals that solve the oracle problem to some extent, and many of them aim to the oracle automation for many kinds of applications.

[Aggarwal et al. \(2004\)](#), [Jin et al. \(2008\)](#) and [Sangwan et al. \(2011\)](#) used ANN (Artificial Neural Networks) acting as an oracle of an SUVV that provides the classification of triangles (equilateral, isosceles, scalene or not triangle) by analyzing the relationship between its three sides. Although the example is not complex, the authors proposed the use of ANN for complex problems that have no analytical solution.

Boyapati et al. (2002) presented *Korat*, a framework in Java for generating and executing test cases, and creating oracles from specifications in JML (Java Modeling Language). The tool translates postconditions in JML to assertions in Java. If SUVV execution violates any assertion, an exception is sent identifying a violated postcondition. The oracle accumulates these exceptions and reports violations.

Chen et al. (2003) proposed a combined solution that integrates metamorphic with fault-based testing using real inputs and symbolic executions. As in Singh et al. (2011), the focus is on performing validation tests without the need to obtain a test oracle, and the major limitation of this methodology is the difficult to found such relations. Based on this work, Sun et al. (2011) used the metamorphic approach for testing web services, and Murphy et al. (2009) implemented a metamorphic solution by using JML specifications which contain metamorphic relations properties.

Harman et al. (2010) addressed the oracle cost, rather than its coverage. They mention that the only way to reduce the oracle cost is by reducing the number of generated tests and, for this objective, present three algorithms that were used in five SUVV examples demonstrating that is possible to reduce the number of test cases without any substantial coverage loss. The problem of cost reduction in test execution also has been approached by Santiago et al. (2008b).

Memon and Xie (2005) presented a technique for test oracle generation for GUI, determining that GUI behaves as specified for certain test cases. The oracle is constructed by means of a specific formal model developed for the project. The oracle derives the expected outputs based on this model, and obtains the current output by an execution monitor. An automatic checker compares results and determines whether GUI is behaving properly or not.

Nardi (2014) considered that generating fully-automated test oracles is unfeasible and proposes an intermediate solution that partially automates test oracle generation for embedded systems, once it is represented by Simulink-like models. As occurs in most of model-driven testing, a great challenge is the manual effort spent to build a model that represents the SUVV and the mapping between model and specification.

LFA (Log File Analysis) (TU et al., 2009) techniques were used in order to generate test oracles. A log analyzer is specified by means of an FSM and a parser converts this representation into a Java program. This approach has some limitations, especially with respect to the difficulty in specifying recursive concepts and more complex



analyzers that require a lot of states to be represented directly in FSM.

Wang et al. (2011) investigated the oracle construction for reactive systems without explicit specifications and making use of ML (Machine Learning) techniques. The InTOL (Intelligent Test Oracle Library) tool, that gathers traces of system execution when their calls are placed in the SUVV, was presented. These execution traces are used to train a SLA (Supervised Learning Algorithm) which derives the oracle.

### 2.5.3 Considerations of Related Literature

Through the analysis of the related literature in Section 2.5, it is possible to consider that there are really certain types of defects that may be detected by means of static analysis. Furthermore, several studies show that the combination of static analysis and dynamic information significantly increases the efficacy of tests for detection of defects, rather than using only static analysis. The analysis with dynamic information is based on real information from SUVV behavior, discarding impossible situations and focusing more on the paths exercised by test cases. REACTOR has a combination of both theories, test oracle and static analysis, and thus it presents the benefits of both methods in a single approach. Moreover, REACTOR tool has a high degree of automation and does not need any extra effort from developers beyond the source code.

Several approaches were already proposed to solve the oracle problem. However, none of them achieved one of the most important aspect, which is a full or really significant automation of this process. In general, these researches presented approaches which needed the manual build of specifications or models, or training of neural networks. In cases where the level of automation is slightly larger, it is observed that they were only tested with simple case studies.

Many related work mentions that the system requirements documentation is the most precious resource when trying to reveal faults (PETERS; PARNAS, 1998), and one shall say that the tests tends to be as good as the documentation associated with SUVV is.

Text documents written in NL still is the most common method for specifying software requirements. However, NL is not formal, and it is therefore very susceptible to several issues of incompleteness, inconsistency or ambiguity (SANTIAGO JÚNIOR, 2011).

Due to the difficulty in handling and interpreting NL, most of related and cited

research studies use an “intermediate” formalism. It is obvious that the adoption of a formal syntax and semantics, brings a great advantage enabling greater ease of developing automated tools. However, one must consider that in many cases the cost of maintaining updated software models (mostly for complex software) can be prohibitive. That is the reason why this thesis is engaged to research testing methodologies that do not require documents or any other formalisms.

## **2.6 Final Remarks**

This chapter presented the basic terminology related to testing techniques commented in this work, and introduces several testing techniques which can be applied on SUVV as black box, white box and model-based. Automated tests provides several advantages in efficiency for testing. Also, it was discussed the importance of reverse engineering for software testing and techniques that were used to generate test data. The static source code analysis can detect defects and extracting information about the internal structure of an SUVV by reverse engineering. Several researches aims to use static and dynamic approaches as complementary techniques in order to let the most accurate defect detection. The inference of expected results is the more complex task to implement a test oracle generator, and this the why there is a lot of published research focusing to solve the oracle problem with certain degree of automation.

### 3 THE REACTOR METHOD

The REACTOR method (ARANTES et al., 2015) combines static code analysis, test case generation and test oracle supported by reverse engineering in order to detect software defects considering only the source code. In this chapter, we present the main concepts, algorithms of the REACTOR method as a general scientific contribution. In other words, although we developed a tool, REACTOR and whose implementation details are in Chapter 4, most of the contributions of REACTOR are not specific to a particular programming language. As we mentioned in Chapter 1, case studies were developed in Java, but most of the ideas behind REACTOR can be adapted to other programming languages such as C++ or C#.

The REACTOR method is shown in Figure 3.1. The source code example of *factorial problem* (FACTORIAL, 2015) in Figure 3.2 is used in order to illustrate how the method works.

Figure 3.1 - The REACTOR method workflow.

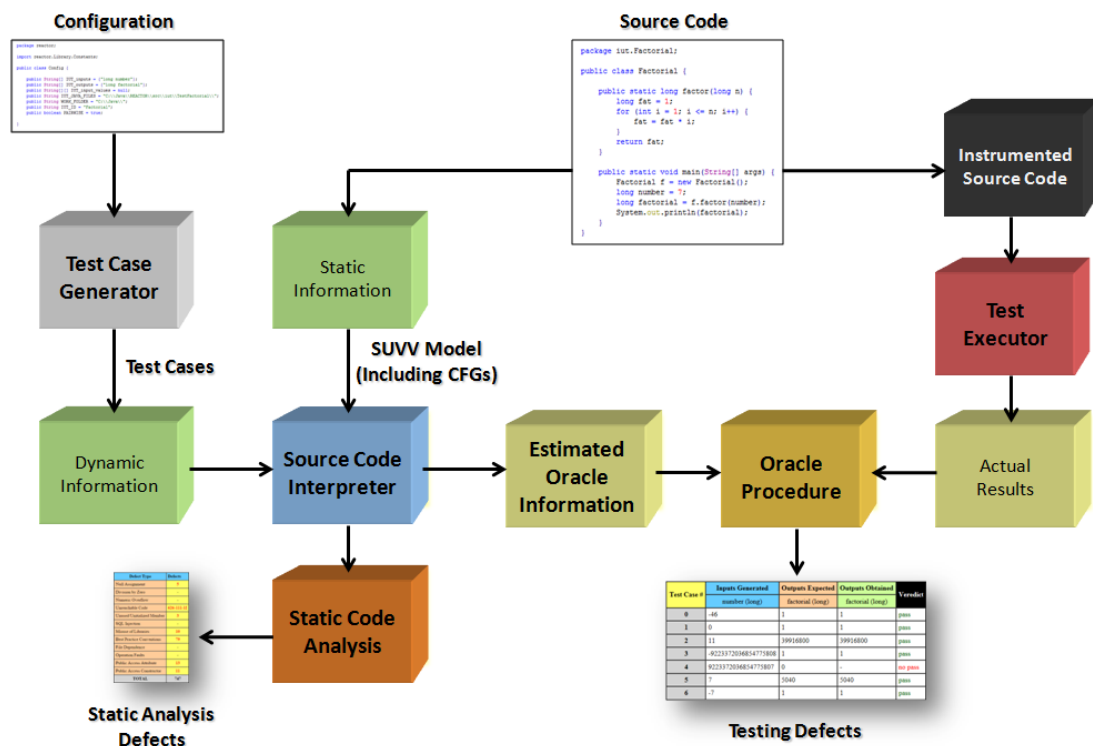


Figure 3.2 - Source code of *factorial problem*.

```
1 package iut.TestFactorial;
2
3 public class Factorial {
4
5     public long factor(long n) {
6         long fat = 1;
7         for (int i = 1; i <= n; i++) {
8             fat = fat * i;
9         }
10        return fat;
11    }
12
13    public static void main(String[] args) {
14        Factorial fat = new Factorial();
15        long number = 7;
16        long factorial = fat.factor(number);
17        System.out.println(factorial);
18    }
19 }
```

### 3.1 Dynamic Information via Test Case Generation

A test case is usually defined as a set of test input data and their respective expected results. However, in some cases, it may consider a test case only as the test input data generated by the approaches, being the expected results omitted. Thus, specifically for this work, when “test case” is mentioned, it means only the test input data. Therefore, in REACTOR a test case is as a set of primitive variables (inputs) that may be used to SUVV testing.

In REACTOR, test cases are automatically generated by a combination of three black box techniques (BVA, EP, RT) which looks at the types of input variables that the tester should configure based on her/his knowledge. Such information, and also other as the directory containing SUVV must be set in attributes coded in a configuration class of REACTOR. Moreover, pairwise testing (LEI; TAI, 1998) is used to decrease the size of the test suite.

In *factorial problem*, test cases were generated based on its input variable which is a long type value, and the expected result is also a long type value (as can be seen in Figure 3.3).

Figure 3.3 - Input and output of *factorial problem* source code.

```
1 package iut.TestFactorial;
2
3 public class Factorial {
4
5     public long factor(long n) {
6         long fat = 1;
7         for (int i = 1; i <= n; i++) {
8             fat = fat * i;
9         }
10        return fat;
11    }
12
13    public static void main(String[] args) {
14        Factorial fat = new Factorial();
15        long number = 7;
16        long factorial = fat.factor(number);
17        System.out.println(factorial);
18    }
19 }
```

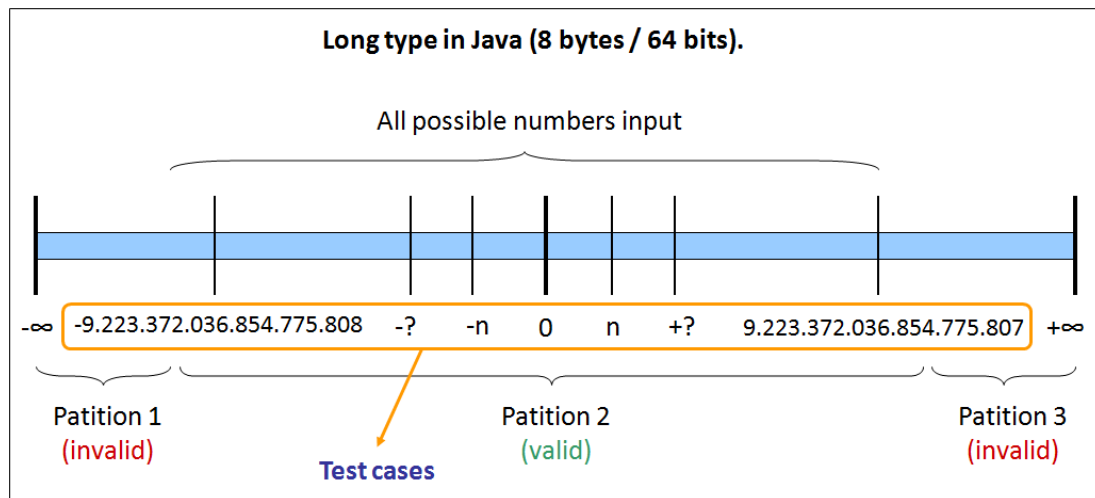
### 3.1.1 Automatic Equivalent and Boundary Partition Analysis

Since it was unable to find a known technique for generating test cases that is perfectly adaptable to our approach, it was necessary to create an *ad hoc* technique suitable for this work. This was done by adapting three known techniques in a way that they could complement each other, in order to provide a consistent and coherent set of values to be used when testing an SUVV in normal and extreme conditions. So, for the purpose of this work, BVA, EP and RT were fused in a technique that was named AEBPA (Automatic Equivalent and Boundary Partition Analysis), which is designed to address more especially exception handling defects that become failures.

AEBPA merges the idea of testing the boundary values suggested by BVA and with range partitions of EP, and adds the RT (GODEFROID et al., 2005) that is recommended by several authors.

Figure 3.4 illustrates how AEBPA selects values for testing, in case of range of integer values in Java. So, in AEBPA, the test case is composed by the lower and higher limits for the variable type, zero, a negative and positive fixed values (represented by  $-n$  and  $n$ ), and a negative and positive random values (represented by  $-?$  and  $?$ ). The fixed values can be coded by the tester, and random values are automatically generated on the fly.

Figure 3.4 - Automatic Equivalent and Boundary Partition Analysis for integer type variables.



It is interesting to mention the differences between classic techniques (BVA, EP and RT), and their principles implemented in AEBPA. In AEBPA, note that the range of values was divided in three partitions. Values above the higher limit, and below the lower limit, are considered invalid. Hence, values within these partitions are not tested. This is a little different to BVA, where a value just below the lower limit and a value just above the upper limit are also tested, but it makes sense computationally, since it is not plausible to expect that an integer stores a greater (or lower) value than allowed by its memory limit without generating a failure.

It also modifies the idea of EP, since in classic approach a negative, zero and positive value can be within the same partition. And consequently, the test with only one value from each partition would be considered a good test even when dealing with such different values. In AEBPA negative, zero, and positive values are always tested even if they are within the same partition. Considering as a running example the *factorial problem* (FACTORIAL, 2015), Table 3.1 presents the list of values used to generate test cases automatically.

Table 3.1 - Test cases generated for the *factorial problem*.

#	Value (long)	Justification
1	-9223372036854775808	Lower boundary for long type
2	-94	Random negative value
3	-7	Fixed negative value from equivalence class
4	0	Zero
5	7	Fixed positive value from equivalence class
6	49	Random positive value
7	9223372036854775807	Upper boundary for long type

It is obvious that the largest number of values to be tested by AEBPA increases the number of test cases, and this can make the test unfeasible especially if there are many input variables that have their values combined. Therefore, a method would be very welcoming to reduce the number of redundant test cases following a particular criteria.

Combinatorial designs have been used as a means to generate shorter and more efficient test suites (MATHUR, 2008). These techniques have been found to be effective in the discovery of faults (defects) due to the interaction of various input variables (BALERA; SANTIAGO JÚNIOR, 2015).

### 3.1.2 Reduction of Test Cases by Pairwise Testing

REACTOR implements an *ad hoc* pairwise that works once SUVV has three or more input variables (it is not applicable for *factorial problem*). So, it reads a test suite with all test case combinations that were exhaustively generated by AEBPA, and then creates a new test suite (that overwrites the old one) with a reduced number of test cases. This new test suite covers all combinations of two, therefore, are much smaller than exhaustive old ones. And still very effective in finding defects. So, the pairwise algorithm is based on the analysis of test suite, defined as follows:

$$suite = \{ tuple \parallel tuple = F, IV, TC, RTC \}$$

$$IV = \{ IV_c \parallel IV_c : input\ values \}$$

$$TC = \{ TC_i \parallel TC_i : test\ cases\ set \}$$

$$RTC = \{ RTC_k \parallel RTC_k : reduced\ test\ cases\ set \}$$

Algorithm 1 presents the pairwise testing, where it should be considered that *suite*

is a tuple which is composed by three elements.  $IV_c$  is a reference to the input values generated by AEBPA.  $TC_i$  is a reference to the set of test cases which compose the test suite, and  $RTC_k$  is a reference to the reduced set of test cases generated by pairwise algorithm.

The input values of all input variables (generated by AEBPA) are concatenated in a set that is counted (lines 3 to 5), and all possible combinations that can be obtained by arranging these values in different orders is stored in  $ts$  (line 6). According to the order of input variables, all impossible combinations are eliminated from  $ts$  (line 7) and all possible combinations of pair positions are stored in  $pairs$  (line 8). Then, each test case is analyzed by comparing with all other test cases (line 9) by searching all its possible pairs of values in other test cases (lines 10 to 23). If all pairs of values from a particular test case are not already set in other test cases (line 27), this test case is added to the reduced test case set ( $RTC$ ) in line 29.

Table 3.2 shows as an example, a test case suite exhaustively generated from a hypothetical system that has three possible inputs:  $A$ ,  $B$  and  $C$ . Table 3.3 shows the same test suite generated by the pairwise algorithm implemented in REACTOR. Note that, even keeping all combined pairs covered, the pairwise reduced the set of test cases from 27 to only 12.

To demonstrate how pairwise can reduce the set of test cases, consider the test case 11. This test case is set by values  $B$ ,  $C$  and  $B$ . So, all possible combinations of two values are  $B$  and  $C$  as the first and second value, that is already covered by test case 7,  $B$  and  $B$  as the first and third value, that is already covered by test case 8, and  $C$  and  $B$  as the second and third value, that is already covered by test case 3. The same occurs with test case 21 which is set by values  $C$ ,  $A$  and  $A$ . The possible combinations of two values are  $C$  and  $A$  as the first and second value, that is already covered by test case 10,  $C$  and  $A$  as the first and third value, that is already covered by test case 12, and  $A$  and  $A$  as the second and third value, that is already covered by test case 6. Therefore, once previous test cases already cover all pairs of test cases 11 and 21, these test cases can be discarded.

In order to demonstrate the effectiveness of pairwise capacity, Table 3.4 shows its reduction rate for several number of possible inputs, by comparing the quantity of exhaustively generated test cases with the quantity of pairwise reduced test cases. Note that, once that the number of possible inputs increases (from 3 up to 12), the pairwise increases its efficiency by reducing the set of test cases from 44,44% until approximately 15% of the original set when handling 12 possible inputs.



**Input:** Test Suite  
**Output:** Reduced Test Suite

```

1   $ts \leftarrow$  Test Suite;
2   $c \leftarrow 0$ ;
3  for  $IV_c \in ts$  do
4  |    $c \leftarrow c + 1$ ;
5  end
6   $ts \leftarrow$  generate_combinations( $c$ );
7   $ts \leftarrow$  eliminate_impossible_combinations( $ts$ );
8   $pairs \leftarrow$  generate_pairs( $c$ );
9  for  $TC_i \in ts$  do
10 |   for  $P \in pairs$  do
11 |   |    $p1 \leftarrow$  get_first_value( $P$ );
12 |   |    $p2 \leftarrow$  get_second_value( $P$ );
13 |   |    $repeated\_pair \leftarrow$  false;
14 |   |   for  $TC_j \in ts$  do
15 |   |   |   if  $i \neq j$  then
16 |   |   |   |   if  $TC_i[p1] \in TC_j \wedge TC_i[p2] \in TC_j$  then
17 |   |   |   |   |    $repeated\_pair \leftarrow$  true;
18 |   |   |   |   end
19 |   |   |   end
20 |   |   end
21 |   |   if  $repeated\_pair = true$  then
22 |   |   |    $set\_repeated\_pair(P)$ ;
23 |   |   end
24 |   end
25 |   /* check if all pairs are repeated in at least one test case */
26 |    $k \leftarrow 0$ ;
27 |   if  $all\_repeated\_pairs(pairs) = false$  then
28 |   |   /* add  $TC_i$  to reduced test suite */
29 |   |    $RTC_k \leftarrow TC_i$ ;
30 |   |    $k \leftarrow k + 1$ ;
31 |   else
32 |   |   /*  $TC_i$  is eliminated */
33 |   end
34 end

```

**Algoritmo 1:** Algorithm of the pairwise testing.

Table 3.2 - Example of a test case suite exhaustively generated.

Test Case #	Input 1	Input 2	Input 3
1	A	B	C
2	A	B	A
3	A	B	B
4	A	C	A
5	A	C	B
6	A	C	C
7	A	A	B
8	A	A	C
9	A	A	A
10	B	C	A
11	B	C	B
12	B	C	C
13	B	A	B
14	B	A	C
15	B	A	A
16	B	B	C
17	B	B	A
18	B	B	B
19	C	A	B
20	C	A	C
21	C	A	A
22	C	B	C
23	C	B	A
24	C	B	B
25	C	C	A
26	C	C	B
27	C	C	C

Table 3.3 - Example of a test case suite reduced by pairwise.

Test Case #	Input 1	Input 2	Input 3
<b>1</b>	A	B	A
<b>2</b>	A	B	B
<b>3</b>	A	C	B
<b>4</b>	A	C	C
<b>5</b>	A	A	C
<b>6</b>	A	A	A
<b>7</b>	B	C	A
<b>8</b>	B	A	B
<b>9</b>	B	B	C
<b>10</b>	C	A	B
<b>11</b>	C	B	C
<b>12</b>	C	C	A

Table 3.4 - Reduction rate of pairwise test cases.

# of Possible Inputs	Exhaustive Test Cases	Pairwise Test Cases	Reduction Rate
<b>3</b>	27	12	44, 44%
<b>4</b>	64	24	37, 5%
<b>5</b>	125	40	32%
<b>6</b>	216	60	27, 77%
<b>7</b>	343	84	24, 49%
<b>8</b>	512	112	21, 88%
<b>9</b>	729	144	19, 75%
<b>10</b>	1000	180	18%
<b>11</b>	1331	220	16, 53%
<b>12</b>	1728	264	15, 28%

### 3.2 Static Information via Reverse Engineering

Reverse engineering of the source code is an important asset for REACTOR. It feeds the necessary information to REACTOR's static code analysis and testing algorithms to detect defects. So, reverse engineering is based on the reading the source code contained in all SUVV files, and since the source code has a defined syntax, it was proposed to read and understand it using regular expressions, defined as follows:

$$files = \{ tuple \parallel tuple = F, F^r, F^f, F^{dec}, F^{cfg} \}$$

$$F = \{ F_i \parallel F_i : source\ code\ file \}$$

$$F^r = \{ F_i^r \parallel F_i^r : source\ code\ read \}$$

$$F^f = \{ F_i^f \parallel F_i^f : source\ code\ formatted \}$$

$$F^{dec} = \{ F_i^{dec} \parallel F_i^{dec} : source\ code\ decomposed \}$$

$$F^{cfg} = \{ F_i^{cfg} \parallel F_i^{cfg} : source\ code\ cfg \}$$

The complete process that generates SUVV model is presented in Algorithm 2, where it should be considered that *files* is a tuple which is composed by five elements.  $F_i$  is a reference of the source code in file system.  $F_i^r$  is a reference for the contents of source code file read by REACTOR and  $F_i^f$  is a reference for the same source code already formatted.  $F_i^{dec}$  is a reference of the decomposed source code and, finally,  $F_i^{cfg}$  is the decomposed source code arranged in CFG.

Depending on the SUVV, REACTOR must be properly configured with essential data which are loaded in line 1 and used to generate test cases (line 10). If the pairwise is activated (line 11) and SUVV has more than two input variables (line 12), the set of test cases may be reduced (line 13). So, the source code files of SUVV is grouped in a set (line 17) whose elements are addressed in a loop (line 18). Then the source code of each file is formatted (line 21), decomposed (line 23), and its control structures are arranged in CFG (line 24). The source code is interpreted in line 28, and its defects detected are inserted in a list (line 30), which is used to generate the text file and table (lines 32 and 34). And finally, the SUVV is instrumented (line 36) and saved in file system (line 38).

**Input:** SUVV and Configuration

**Output:** List of Defects and Test Oracle Information

```
1 config ← REACTOR Configurations;
2 suvv ← SUVV Model;
3 dl ← Defect List;
4 fw ← File Writer;
5 if exists a directory of report files then
6 | delete all files;
7 else
8 | creates a new directory;
9 end
10 suvv ← generate_test_cases(suvv);
11 if pairwise is activated then
12 | if SUVV has more than two inputs then
13 | | suvv ← pairwise_testing(suvv);
14 | end
15 end
16 /* search source code files */
17 files ← search_files(config);
18 for  $F_i \in files$  do
19 | /* get contents of source code files */
20 |  $F_i^r$  ← file_reader( $F_i$ );
21 |  $F_i^f$  ← format_source_code( $F_i^r$ );
22 | /* source code decomposition */
23 |  $F_i^{dec}$  ← Call Algorithm 3( $F_i^f$ );
24 |  $F_i^{cfg}$  ← generate_CFG( $F_i^{dec}$ );
25 | suvv ←  $F_i^{cfg}$ ;
26 end
27 /* source code interpretation */
28 Call Algorithm 4(suvv);
29 /* detection of defects */
30 dl ← check_overall_defects(suvv);
31 /* saving text file */
32 save_defect_text(fw , dl);
33 /* saving HTML table */
34 save_defect_table(fw , dl);
35 /* instrumentation of SUVV */
36 new_suvv ← instrument_SUVV(suvv , config);
37 /* saving similar SUVV */
38 save_instrumented_SUVV(fw , new_suvv);
```

**Algoritmo 2:** REACTOR's main algorithm.

### 3.2.1 Regular Expressions

In computer science, a regular expression is a planned sequence of characters that specifies a search pattern, generally for use by matching with string's patterns, similar to “search and replace” operations (LORENZO et al., 2013). Regular expression is usually abbreviated as “*regex*”, and originated from formal language theory.

Regular expressions are commonly applied for text processing utilities, since they are useful to provide both a basic and extended standard for the grammar and syntax. Regular expression processors are available in several applications: search engines, text editors, search and replace dialogs of several text processors, and command line utilities from operational systems. In addition, most of the programming languages provide regular expression functions, some built-in, for example Perl or Ruby, and others via a standard library, for example Java, Python and C++. Many other languages implement regular expressions by means of additional libraries.

REACTOR implements several *regex* patterns that match key code lines which guide the construction of the SUVV model in Figure 3.7, and many other patterns that are responsible to interpret code lines after the decomposition process.

In REACTOR there are basically two types of *regex*: full line and inline *regex*. Full line *regex* matches perfectly with a full source code line without no more decompositions, and is used also to classify a code line. On the other hand, inline *regex* matches with just a part of a code line, and it is used to match and decompose a code line. The decomposition of code lines is more detailed in Section 3.2.4.

Table 3.5 shows some of the major expressions that are used to identify and classify specific standards of full source code lines, and Table 3.6 shows some of the major expressions that are used to identify specific standards of inline source code. All regular expressions are shown in Appendix A.1 (Tables A.1 and A.2). In both tables, the first column describes the code line standard that *regex* matches, and the second and third columns shows, respectively, the *regex* implemented and an example of a code line that it matches.

Table 3.5 - Full code line *regex* used in REACTOR.

Description	Regex	Recognized Standard
Variable assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] ; [ ] \$</code>	<code>var = otherVar ;</code>
Unary operation	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] {0,1} ( -   -   \\\  +   \\\  +   ~   ! ) [ ] ; [ ] \$</code>	<code>i ++ ;</code>
Class declaration	<code>^(public  private  protected ) {0,1} (final ) {0,1} (abstract ) {0,1} (class  interface ) [a-zA-Z_][a-zA-Z0-9_]* [ ] * [ ] ( ( (extends) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] * [ ] {0,1} ) * ) * [ ] {0,1} ( (implements) [ ] ( [a-zA-Z0-9_ \\\  &lt; \\\  &gt; ] * [ ] {0,1} [ ] , [ ] * [ ] {0,1} ) * ) * [ ] {0,1} ( \\\  { } ) \$</code>	<code>public class CodeMatcher extends Matcher {</code>
Method declaration	<code>^(public  private  protected ) {0,1} (static ) {0,1} (final ) {0,1} (abstract ) {0,1} (synchronized ) {0,1} ([a-zA-Z_][a-zA-Z0-9_]* [ ] {0,1} [ \\\  &lt; \\\  &gt; \\\  \\\  ] {0,1} [ ] {0,1} [a-zA-Z0-9_]* [ ] {0,1} [ \\\  &lt; \\\  &gt; \\\  \\\  ] {0,1} [ ] {0,1} [ ] {0,1} [a-zA-Z0-9_]* [ ] {0,1} [ \\\  ( [ ] * ( [a-zA-Z_ - ] [a-zA-Z0-9_ \\\  \\\  ] &lt; \\\  &gt; ] * [ ] [a-zA-Z_][a-zA-Z0-9_ \\\  \\\  ] &lt; \\\  &gt; , ] * ) * \\\  ) [ ] {0,1} (throws [a-zA-Z0-9_]* ( \\\  { } ) \$</code>	<code>public void execute() throws IOException {</code>





In order to demonstrate how *regex* works, let us consider the first example in Table 3.5 which matches the variable assignment. The “`^`” character indicates that this is the beginning of a line, and “`[ ]`” indicates an empty space. There are two expressions that identify a variable name “`[a-zA-Z_][a-zA-Z0-9_]*`” where the first part (“`[a-zA-Z_]`”) means that it must begin with any lowercase/uppercase character or underline. The second part (“`[a-zA-Z0-9_]*`”) is slightly similar with the addition of numbers, since variable names can not begin with numbers but can have numbers after the first character. Also, it has the usual quantifiers “`*`” (zero or more occurrences), “`+`” (one or more occurrences). Finally, the regex ends with “`$`” which indicates the end of line.

By search patterns, detailed static information of the SUVV are recovered by using several regex listed in Table 3.5. Such information are essential to build a model that corresponds the real SUVV, and this model is the basis for the entire operation of REACTOR.

### 3.2.2 Source Code Reading

The first step of REACTOR is the search of source code files of the SUVV. Initially, each of these files is modeled as a class, and its code lines are stored within the model. Only then, each of these files are formatted and analyzed more carefully.

The formatting of code lines means basically by splitting annotations and adjusting spacing. Since the ultimate goal of the REACTOR is to prevent failures, it does not make sense to analyze annotation lines knowing that they do not influence the execution of SUVV. Furthermore, the maintenance of annotations can significantly increase the number of code lines that may be analyzed by *regex* when dealing with well annotated software, so it is an unnecessary processing that can be avoided.

REACTOR defined an annotation technique which is concatenated at the end of each code line in order to delimit scopes (for classes, methods and constructors), and to point out package declarations and import statements, as illustrated in Figure 3.5. Note the annotations `/*%POINT_PACKAGE%*/` which identifies a package, `/*%POINT_CLASS_OPEN#2%*/` identify the opening of a class, and `/*%POINT_METHOD_OPEN#4%*/` identifies a method, among others. Note also that some annotations have a “n” at the end, this character is changed by an automatically incremented number that is used to identify each element of static analysis. The set of annotations used by REACTOR with its respective functions are listed in Table 3.7.

Figure 3.5 - Scopes annotated in *factorial problem* source code.

```

1 package iut.TestFactorial ; /*%POINT_PACKAGE%*/
2
3 public class Factorial { /*%POINT_CLASS_OPEN#2%*/
4
5     public long factor ( long n ) { /*%POINT_METHOD_OPEN#4%*/
6         long fat = 1 ;
7         for ( int i = 1 ; i <= n ; i++ ) {
8             fat = fat * i ;
9         }
10        return fat ;
11    } /*%POINT_METHOD_CLOSE%*/
12
13    public static void main ( String [ ] args ) { /*%POINT_MAIN_OPEN%*/
14        Factorial fat = new Factorial ( ) ;
15        long number = 7 ;
16        long factorial = fat.factor ( number ) ;
17        System.out.println ( factorial ) ;
18    } /*%POINT_MAIN_CLOSE%*/
19 } /*%POINT_CLASS_CLOSE%*/

```

Table 3.7 - Annotations to Delimit Scopes, Packages and Imports.

#	Annotation	Function
0	/*%POINT_PACKAGE%*/	Java Package
1	/*%POINT_IMPORT%*/	Library Import
2	/*%POINT_CLASS_OPEN#n%*/	Class Opening
3	/*%POINT_CLASS_CLOSE%*/	Class Closing
4	/*%POINT_MAIN_OPEN%*/	Main Method Opening
5	/*%POINT_MAIN_CLOSE%*/	Main Method Closing
6	/*%POINT_METHOD_OPEN#n%*/	Method Opening
7	/*%POINT_METHOD_CLOSE%*/	Method Closing

Based on the annotations that delimit programming scopes, the source code is modeled by analyzing its main contents as: attributes, classes, methods and constructors. And once a method or constructor is found and its source code is delimited, it is modeled in SUVV as new method. In modeling, constructors are also modeled as methods with (or without) parameters, since the difference between them is subtle. It is interesting to mention that, if there is an inner class, it is split from source code and modeled as a standard class. And if a specific class is implemented with-

out empty constructors, they are automatically modeled. Source code lines that are within the class boundaries and outside the boundaries of methods or constructors, are stored as attributes of the class model. And all source code that is within the boundaries of methods and constructors are stored as source code of the method modeling.

It is also interesting to mention that, although Figure 3.5 suggests that SUVV files are updated with standard annotations, in fact they are not. Because such updates are performed only with the source code that is stored within the SUVV model, and the source code files are never modified by REACTOR. Therefore, the figure illustrates the annotated source code that is modeled and which is not necessarily visible to software tester in SUVV files.

At this point, it was possible to model classes, methods (and parameters), builders and attributes from SUVV. The next step is to analyze the source code of each method or constructor, and identify the involved control structures. In Java, control structures types can be *if*, *for*, *do*, *while*, *switch* and *try*. Code lines related to these structures are identified by using specific *regex* (Table 3.5) and they are annotated following a standardized classification which is detailed in Section 3.2.4. Figure 3.6 illustrates the source code with an annotated control structure (see the annotation `/*##FOR_OPEN#1##*/`). Table 3.8 lists the set of annotations used by REACTOR with its respective functions.

Figure 3.6 - Control structures annotated in *factorial problem* source code.

```
1 package iut.TestFactorial ; /*$POINT_PACKAGE$*/
2
3 public class Factorial { /*$POINT_CLASS_OPEN#2$*/
4
5     public long factor ( long n ) { /*$POINT_METHOD_OPEN#4$*/
6         long fat = 1 ;
7         for ( int i = 1 ; i <= n ; i++ ) { /*##FOR_OPEN#1##*/
8             fat = fat * i ;
9         } /*$FOR_CLOSE#1$*/
10        return fat ;
11    } /*$POINT_METHOD_CLOSE$*/
12
13    public static void main ( String [ ] args ) { /*$POINT_MAIN_OPEN$*/
14        Factorial fat = new Factorial ( ) ;
15        long number = 7 ;
16        long factorial = fat.factor ( number ) ;
17        System.out.println ( factorial ) ;
18    } /*$POINT_MAIN_CLOSE$*/
19 } /*$POINT_CLASS_CLOSE$*/
```

Table 3.8 - Annotations Identify Control Structures.

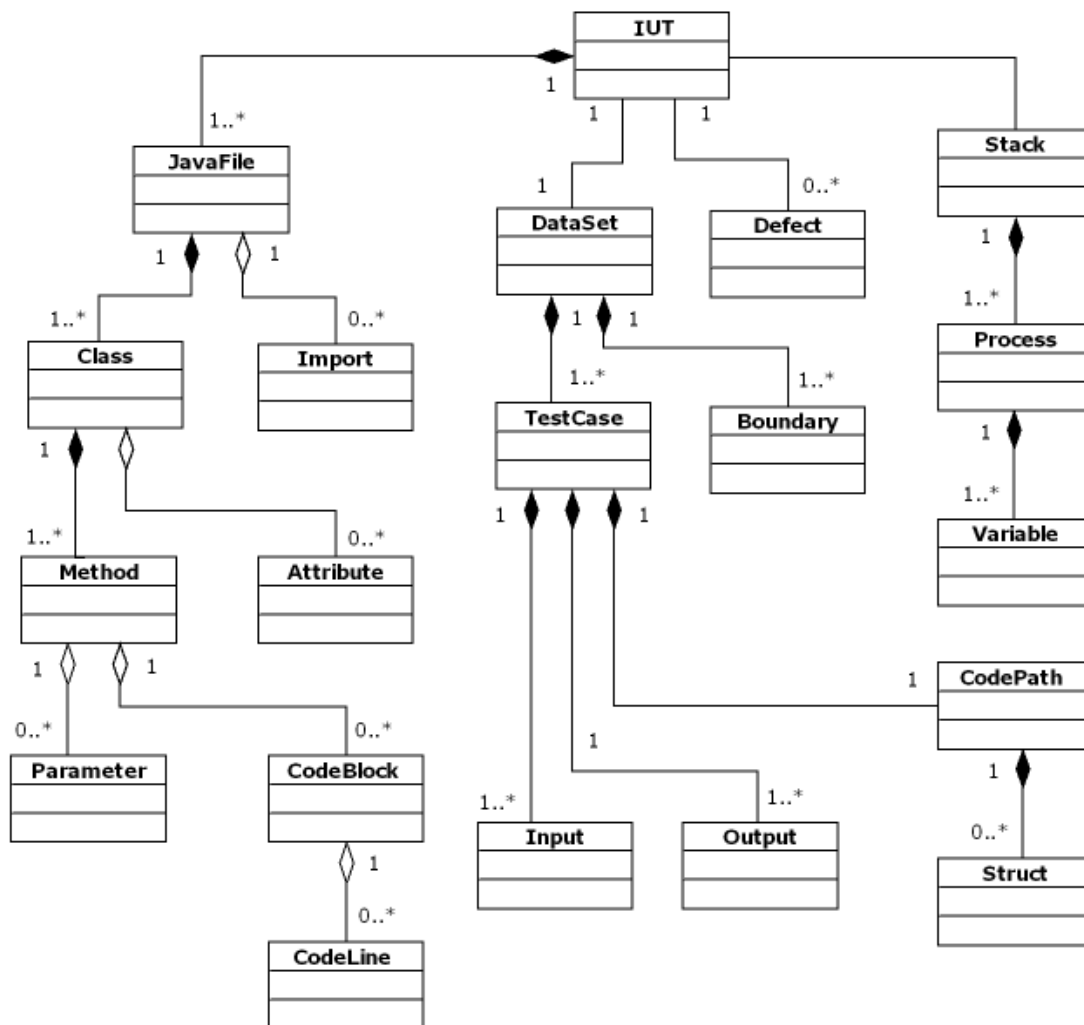
#	Annotation	Function
0	/*#IF_OPEN#n#*/	If Opening
1	/*#IF_ELSE#n#*/	If-Else
2	/*#IF_ELSEIF#n#*/	If-Else-If
3	/*#IF_CLOSE#n#*/	If Closing
4	/*#FOR_OPEN#n#*/	For Opening
5	/*#FOR_CLOSE#n#*/	For Closing
6	/*#WHILE_OPEN#n#*/	While Opening
7	/*#WHILE_CLOSE#n#*/	While Closing
8	/*#DOWHILE_OPEN#n#*/	Do-While Opening
9	/*#DOWHILE_CLOSE#n#*/	Do-While Closing
10	/*#SWITCH_OPEN#n#*/	Switch Opening
11	/*#SWITCH_CASE#n#*/	Switch-Case
12	/*#SWITCH_BREAK#n#*/	Switch-Brake
13	/*#SWITCH_DEFAULT#n#*/	Switch-Default
14	/*#SWITCH_CLOSE#n#*/	Switch Closing
15	/*#BREAK#n#*/	Break Calling
16	/*#CONTINUE#n#*/	Continue Calling
17	/*#TRY#n#*/	Try Opening
18	/*#CATCH#n#*/	Try-Catch
19	/*#FINALLY#n#*/	Try-Finally
20	/*#TRY_CLOSE#n#*/	Try Closing

Figure 3.6 illustrates the source code with a annotated control structure. Note that this annotation also incorporates an integer number that relates a code line with others from the same control structure. As well as with the elements mentioned before (class, methods and constructors), control structures must also be modeled and inserted in SUVV model as CFG. Each method or constructor must be linked to its CFG model.

### 3.2.3 SUVV Modeling

Basically, static information is recovered from source code analysis, such as software elements (classes, methods, and variables) and relationships among them. Relationships can be complex spanning the extension between classes, interfaces, and overwritten or overloaded method calls. A structural model, shown in Figure 3.7, is built based on test cases and source code static information, and this model becomes the basis for detection of defects and test oracle generation.

Figure 3.7 - Structural model of the SUVV created by REACTOR.



It is important to stress that this SUVV model is standardized for every source code that REACTOR assesses. A brief explanation of the meaning of some classes fol-

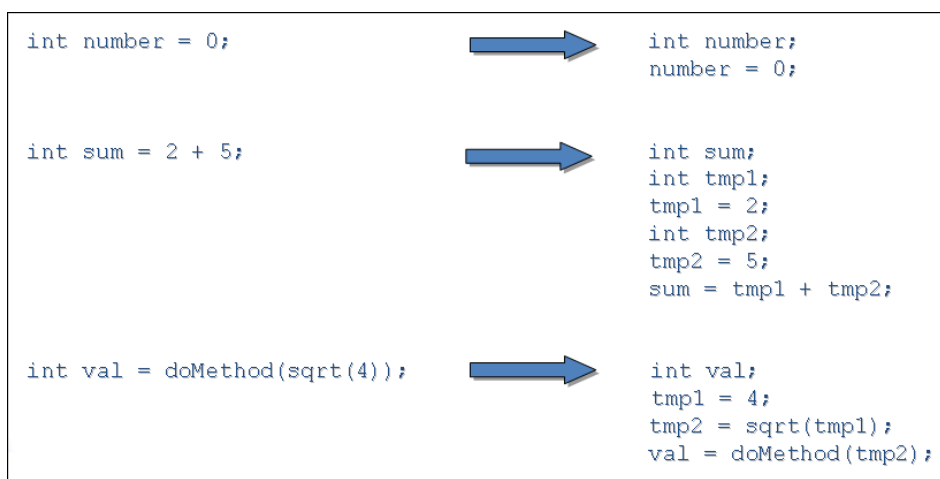
lows. Classes hierarchically below “JavaFile” are based on UML (Unified Modeling Language) (UML, 2015) class diagram, in spite of not following exactly the same standard since it has information that UML does not, such as code blocks and code lines. “TestCase” modeling classes with its input and output values. “Boundary” models the values used in automatic test case generation. “CodePath” and “Structure” represents the path and control structures exercised by a test case. A “Defect” class, such as its name suggests, is the modeling of defects that are detected. “Stack”, “Process”, and “Variable” are used in source code interpretation.

The SUVV modeling requires the insertion of classification annotations in all code lines in accordance with Table 3.5. This is more complex, because in many cases, it requires the decomposition of non-classifiable code lines in classifiable code lines.

### 3.2.4 Source Code Decomposition

The decomposition process is basically breaking each SUVV’s code line into one or more standardized and less complex code lines, since only standardized code lines can be accurately recognized using *regex*. Therefore, each code line may be decomposed into a finite set of simpler and standardized lines for source code representation, so that almost any software can be written using only this set which consists of a finite and relatively small number of code line possibilities. Three examples of this decomposition process are shown in Figure 3.8.

Figure 3.8 - Example of source code decomposition.



The source code decomposition is a required process, because it is possible to write a code line to perform a specific function using many different syntaxes. Therefore, code lines must be decomposed by using syntaxes known and understandable by REACTOR, as defined follows:

$$suvv = \{ tuple \parallel tuple = C , M , CL^c , CL^d \}$$

$$C = \{ C_i \parallel C_i : suvv \text{ class reference} \}$$

$$M = \{ M_i \parallel M_i : suvv \text{ method reference} \}$$

$$CL^c = \{ CL_{ij}^c \parallel CL_{ij}^c : \text{current code line} \}$$

$$CL^d = \{ CL_{ik}^d \parallel CL_{ik}^d : \text{decomposed code line} \}$$

A summarized version of this decomposition process is shown in Algorithm 3, where it should be considered that *suvv* is a tuple which is composed by four elements.  $C_i$  is a reference for the class of SUVV modeled in REACTOR.  $M_i$  is a reference for the method or constructor which belongs to  $C_i$ .  $CL_{ij}^c$  is a set which contains the code lines of SUVV that belongs to  $C_i$  and  $M_i$ . And finally,  $CL_{ik}^d$  contains the decomposed code lines equivalent to the code lines of  $CL_{ij}^c$ .

In Algorithm 3, every code line composes a list of code lines which is linked to its method instance (line 3) and to its respective class instance (line 2) of SUVV. These are read and tested by a set of *regex* parsers which verify if it is possible to classify without the decomposition (line 7). If it is possible, the code line is considered already decomposed and is added to  $CL_{ik}^d$  (lines 8, 9 and 10), otherwise, it may be decomposed. The part of code line that is decomposed is assigned to a temporary variable (line 13) which also replaces the matched region in current code line (line 15), and this loop (lines 12 to 19) is done until the current code line becomes a code line that can be classified in Lines 19, 20 and 21. The SUVV can only be understood by REACTOR once it is decomposed. Note that the “ $\wedge$ ” signal means a concatenation of two strings in lines 9, 17 and 21.

**Input:** SUVV Model

**Output:** Decomposed Code Lines of SUVV

```
1 suvv ← SUVV Model;
2 for  $C_i \in \textit{suvv}$  do
3   for  $M_i \in C_i$  do
4      $j \leftarrow 0$ ;
5      $k \leftarrow 0$ ;
6     for  $CL_{ij}^c \in M_i$  do
7       if  $\textit{isClassifiable}(CL_{ij}^c) = \textit{true}$  then
8          $\textit{ann} \leftarrow \textit{annotate\_codeline}(CL_{ij}^c)$ ;
9          $CL_{ik}^d \leftarrow CL_{ij}^c \frown \textit{ann}$ ;
10         $k \leftarrow k + 1$ ;
11      else
12        while  $\textit{isClassifiable}(CL_{ij}^c) = \textit{false}$  do
13           $\textit{tmpVar} \leftarrow \textit{new\_temporary\_variable}(\textit{suvv})$ ;
14           $\textit{cp} \leftarrow \textit{split\_classifiable\_code\_part}(CL_{ij}^c)$ ;
15           $CL_{ij}^c \leftarrow \textit{replace}(CL_{ij}^c, \textit{cp}, \textit{tmpVar})$ ;
16          /* CLikd is a new decomposed code line */
17           $CL_{ik}^d \leftarrow \textit{tmpVar} \frown "=" \frown \textit{cp} \frown ";"$ ;
18           $k \leftarrow k + 1$ ;
19        end
20         $\textit{ann} \leftarrow \textit{annotate\_codeline}(CL_{ij}^c)$ ;
21         $CL_{ik}^d \leftarrow CL_{ij}^c \frown \textit{ann}$ ;
22         $k \leftarrow k + 1$ ;
23      end
24    end
25  end
26 end
```

**Algoritmo 3:** Algorithm of the source code decomposition.



Returning to the *factorial problem* example, Figure 3.9 demonstrates how the decomposition process acts on *factorial problem* source code by showing differences between original source code and its equivalent decomposed source code. Note that, in this example, there are single code lines that were decomposed in three according to the classification proposed in Section 3.2.5.

Figure 3.9 - Example of source code decomposition of *factorial problem*.

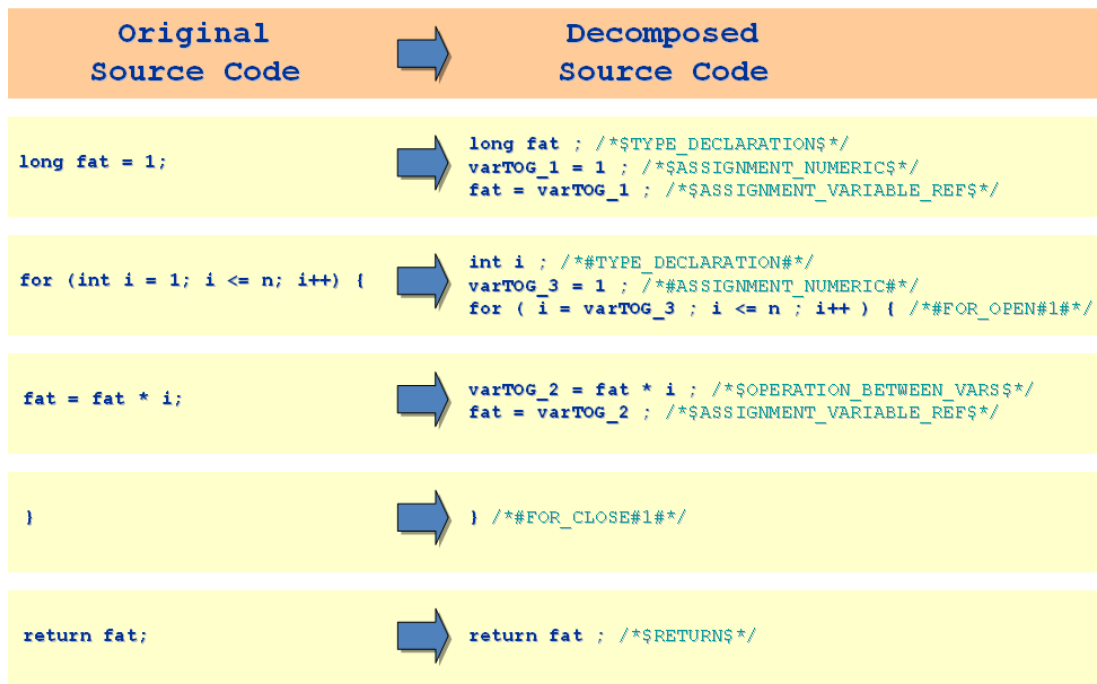


Figure 3.10 illustrates the complete *factorial problem* source code after the decomposition process. It can be seen that the decomposition increased the number of SUVV code lines, but they are now simple and classifiable by the *regex* in Table 3.5.

### 3.2.5 Source Code Classification

As mentioned before, when each code line is identified by *regex* (Table 3.5) and decomposed, they are then classified based on a known set, thus generating another source code easier to be read by a finite set of *regex* parsers, however more complex to be understood by humans, and equivalent to the original SUVV. This known set, shown in Appendix A.2 (Table A.3), is composed of 51 standardized code line types which cover several possibilities of writing a source code in Java. The set of annotations used by REACTOR to identify and classify code lines with its respective

Figure 3.10 - Code lines of *factorial problem* after the decomposition process.

```
1 package iut.TestFactorial ; /*%POINT_PACKAGE%*/
2
3 public class Factorial { /*%POINT_CLASS_OPEN#2%*/
4
5     public long factor ( long n ) { /*%POINT_METHOD_OPEN#4%*/
6         long fat ; /*%TYPE_DECLARATION%*/
7         varTOG_1 = 1 ; /*%ASSIGNMENT_NUMERIC%*/
8         fat = varTOG_1 ; /*%ASSIGNMENT_VARIABLE_REF%*/
9         int i ; /*%TYPE_DECLARATION%*/
10        varTOG_3 = 1 ; /*%ASSIGNMENT_NUMERIC%*/
11        for ( i = varTOG_3 ; i <= n ; i++ ) { /*%FOR_OPEN#1%*/
12            varTOG_2 = fat * i ; /*%OPERATION_BETWEEN_VARS%*/
13            fat = varTOG_2 ; /*%ASSIGNMENT_VARIABLE_REF%*/
14        } /*%FOR_CLOSE#1%*/
15        return fat ; /*%RETURN%*/
16    } /*%POINT_METHOD_CLOSE%*/
17
18    public static void main ( String [ ] args ) { /*%POINT_MAIN_OPEN%*/
19        Factorial fat ; /*%TYPE_DECLARATION%*/
20        fat = new Factorial ( ) ; /*%ASSIGNMENT_NEW_OBJECT%*/
21        long number ; /*%TYPE_DECLARATION%*/
22        varTOG_0 = 7 ; /*%ASSIGNMENT_NUMERIC%*/
23        number = varTOG_0 ; /*%ASSIGNMENT_VARIABLE_REF%*/
24        long factorial ; /*%TYPE_DECLARATION%*/
25        factorial = fat.factor ( number ) ; /*%FUNCTION_RETURN_FROM_REF%*/
26        System.out.println ( factorial ) ; /*%PRINT%*/
27    } /*%POINT_MAIN_CLOSE%*/
28 } /*%POINT_CLASS_CLOSE%*/
```

functions are listed in Table 3.9.

The main reasoning behind this classification is to provide a simple and automatic description of what each code line does. And, based on this description, REACTOR may interpret the code line content to help in the detection of static and testing defects. However, code lines must be interpreted following the sequence provided by their control structures, therefore these structures should be somehow modeled.

Table 3.9 - Annotations to Classify Code Lines.

#	Annotation	Function
0	/*\$TYPE_DECLARATION\$*/	Variable Declaration
1	/*\$ASSIGNMENT_STRING\$*/	String Value Assignment
2	/*\$ASSIGNMENT_CHAR\$*/	Char Value Assignment
3	/*\$ASSIGNMENT_NUMERIC\$*/	Numeric Value Assignment
4	/*\$ASSIGNMENT_BOOLEAN\$*/	Boolean Value Assignment
5	/*\$ASSIGNMENT_VARIABLE_REF\$*/	Other Variable Assignment
6	/*\$ASSIGNMENT_HEXADECFIMAL\$*/	Hex Value Assignment
7	/*\$ASSIGNMENT_NULL\$*/	Null Value Assignment
8	/*\$ASSIGNMENT_NEW_OBJECT\$*/	New Object Assignment
9	/*\$ARRAY_DECLARATION\$*/	New Array Declaration
10	/*\$NEW_ARRAY\$*/	New Array Assignment
11	/*\$ASSIGNMENT_FROM_ARRAY_ELEMENT\$*/	Value From Array Assignment
12	/*\$ASSIGNMENT_TO_ARRAY_ELEMENT\$*/	Value To Array Assignment
13	/*\$OPERATION_UNARY_LEFT\$*/	Unary Operation (Left Side Operator)
14	/*\$OPERATION_UNARY_RIGHT\$*/	Unary Operation (Right Side Operator)
15	/*\$OPERATION_BETWEEN_VARS\$*/	Operation Of Two Variables
16	/*\$FUNCTION_CALL\$*/	Method Calling
17	/*\$FUNCTION_FROM_REF\$*/	Method Calling From Object Reference
18	/*\$FUNCTION_RETURN\$*/	Method Calling With Return Value

(Continue)

Table 3.9 - Continuation

#	Annotation	Function
19	/*\$FUNCTION_RETURN_FROM_REF\$*/	Method Calling From Object Reference With Returned Value
20	/*\$CASTING\$*/	Type Casting Value
21	/*\$PRINT\$*/	Print Variable Value
22	/*\$RETURN\$*/	Return Variable Value
23	/*\$THROW_EXCEPTION\$*/	Throw Running Exception
24	/*\$LISTCLASS_DECLARATION\$*/	Java Collection Declaration
25	/*\$NEW_LISTCLASS\$*/	New Java Collection
26	/*\$PUBLIC_ATTRIBUTE\$*/	Public Attribute of Class
27	/*\$THIS_ASSIGNMENT\$*/	This Reference Assignment
28	/*\$SUPER_ASSIGNMENT\$*/	Super Reference Assignment

### 3.2.6 CFG Generation

Internal control structures of a source code must be modeled, in order to guide the correct interpretation of code lines. And the most appropriate approach for this purpose is by arranging classified code lines in blocks and connect these blocks similar to a CFG (MARINKE, 2012) model.

In this work, CFG is adapted to represent each method within the SUVV assuming that graph nodes are the source code blocks (i.e., a finite list of code lines) and the edges are the connections between them due to control structures in Java, such as: *if-else*, *for*, *while*, *do-while*, *switch-case*, and *try-catch*. Thus, the CFG is implemented by classes “CodeBlock” and “CodeLine” in Figure 3.7. Each method will have a related CFG and thus the SUVV model will have as many CFGs as the number of methods of all SUVV’s classes.

Figure 3.11 shows the *factorial problem* source code properly decomposed, classified and arranged into code blocks. Note that there is a sequence of blocks connected within *factor* method and the *main* method has only one block, since it does not have any control structure involved.

Figure 3.11 - Code lines of *factorial problem* arranged in code blocks.

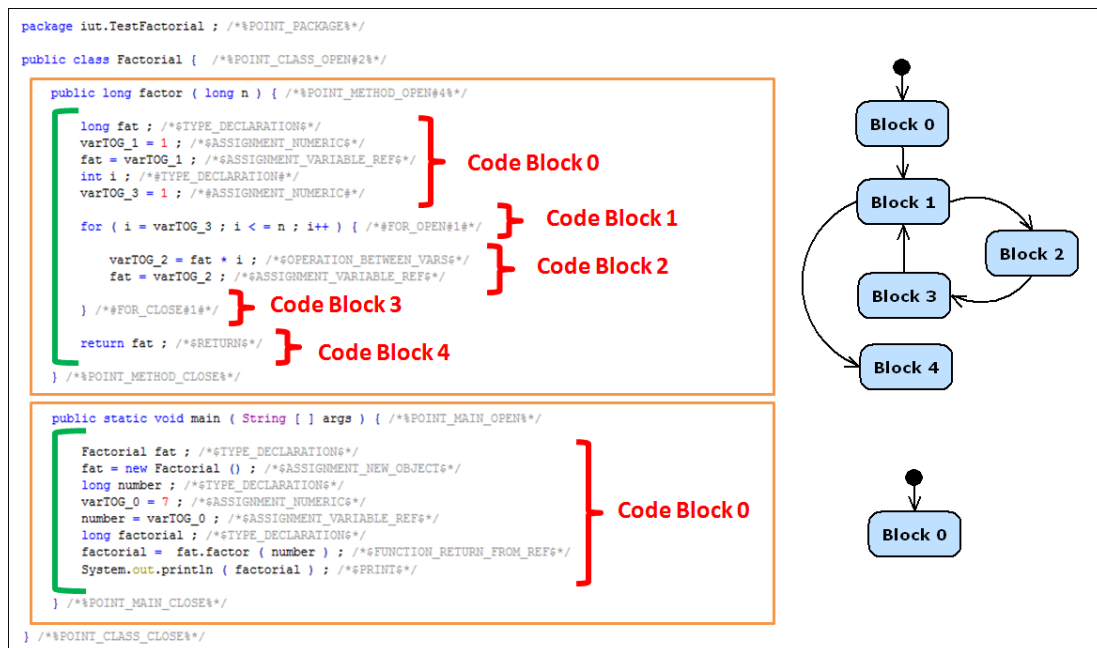


Table 3.10 presents the information of each block that is stored in REACTOR. *block\_id* and *block\_name* are, respectively, the numerical and named identification of the block. *codelines* is an array of code lines (i.e., objects of “CodeLine” class) which stores all code lines related to the code block, and note that its four code lines refers to the first code block of factor method in *factorial problem* decomposed source code. A reference (or pointer) from the “method” object of SUVV model is in *method*. Each code block can be of structural or common source code, and the attribute *type* is set as “s” or “c” depending on this. And finally, the *boolean* value of *isUsed* is set as “true” once this code block is interpreted, that is useless source code many times.

Table 3.10 - Code block of *factor* in REACTOR.

Attribute	Value
int block_id	0
String block_name	“Block0”
ArrayList<CodeLine> codelines	[0]: “long fat ;” [1]: “varTOG <sub>1</sub> = 1;” [2]: “fat = varTOG <sub>1</sub> ,” [3]: “int i ;”
Method method	“object reference”
char type	“c”
boolean isUsed	false

### 3.3 Source Code Interpreter

In this work, static approach is automated and used to provide all information necessary for understanding the SUVV and assembling a model that can be analyzed to reveal defects and provide the estimated expected result. The way that static analysis is combined with dynamic information and estimated results are inferred from the SUVV model is by interpreting of code lines arranged in code blocks. So, for each generated test cases, code blocks are accessed and their code lines are interpreted following the same sequence of a conventional execution, starting from the first code line of the first block. So, the source code is interpreted by a hybrid technique that combines static and dynamic information in order to get additional information that can only be addressed with the observation of SUVV with test

cases. Usually the static code analysis tools do not use such dynamic information, however, in REACTOR the use of dynamic information makes available the values of variables in a certain computation step and it is used as a complementary way in order to detect defects.

The interpretation of decomposed source code is performed by simulating the SUVV behavior with test case inputs, similar to a real execution. A preliminary source code reading is done in order to list every variable declared in SUVV, including temporary variables created by the tool in decomposed source code. In REACTOR, a variable can be a class attribute, local variable or method parameter. Furthermore, in case of class inheritance, the variable must be accessible by other classes that inherited it. A list of all SUVV variables is an attribute of the “Process” class in the SUVV model. This attribute (list) for the *factorial problem* is in Table 3.11.

One of the columns in Table 3.11 is the scope of variable. Every variable listed has its specific scope, which is determined based on its statement location. The specification of a scope is essential to allow variables with equal names in different locations of SUVV. The scope is an auto incremented integer value automatically generated that is the key of a modeled class, method or constructor. In *factorial problem* example, Table 3.12 lists all scopes of SUVV and it can be seen that there is a variable named *fat* that exists in two different scopes (see Table 3.11).

Table 3.11 - Process used in *factorial problem*.

Variable	Scope	Type	Value
<b>n</b>	1	long	-
<b>fat</b>	1	long	-
<b>varTOG_1</b>	1	-	-
<b>i</b>	1	int	-
<b>varTOG_3</b>	1	-	-
<b>varTOG_2</b>	1	-	-
<b>fat</b>	2	Factorial	-
<b>number</b>	2	long	-
<b>varTOG_0</b>	2	-	-
<b>factorial</b>	2	long	-

Table 3.12 - Scopes used in *factorial problem*.

ID	Type	Path	Parameters
0	Class	Factorial	-
1	Method	Factorial.factor	1
2	Method	Factorial.main	0

The interpretation starts with the *main* method by reading code blocks respecting block sequence that may be modified by control structures. Each operation that results in change of a variable value, updates the variable list taking care to ensure that it only updates variables that are within the current scope. When the interpretation has a scope change reading from declaration of a new class or by calling a constructor or method, a new process is created bringing its own new list of variables (process). One for each code line interpreted, and another in the path of code lines exercised for each test case. At the end of interpretation, when the last code line of the *main* method is read, input and output values obtained are captured and stored for the final report. At the end of this step, a new test case is initialized. After simulating all test cases, the final verdicts are determined through the comparison of expected results with results obtained by executing a test implementation that is automatically generated.

The interpretation infers the expected result of a test case by reading code blocks respecting the sequence dynamically determined by control structures. The summarized approach of the source code interpreter is presented by Algorithms 4, 5 and 6. Note that Algorithms 5 and 6 are subroutines of 4. The set of information used by these algorithms is defined follows:

$$dataset = \{ tuple \parallel tuple = TC, P, DD, IN, IV, ON, OV, VLN, VLV \}$$

$$TC = \{ TC_i \parallel TC_i : suvv \text{ test case reference} \}$$

$$P = \{ P_i \parallel P_i : \text{test case path} \}$$

$$DD = \{ DD_i \parallel DD_i : \text{list of defects detected} \}$$

$$IN = \{ IN_{ij} \parallel IN_{ij} : \text{test case input names} \}$$

$$IV = \{ IV_{ij} \parallel IV_{ij} : \text{test case input values} \}$$

$$ON = \{ ON_{ik} \parallel ON_{ik} : \text{test case output names} \}$$

$$OV = \{ OV_{ik} \parallel OV_{ik} : \text{test case output values} \}$$



$$\begin{aligned}
VLN &= \{ VLN_{il} \parallel VLN_{il} : \text{variable list names} \} \\
VLV &= \{ VLV_{il} \parallel VLV_{il} : \text{variable list values} \}
\end{aligned}$$

There are two main sets handled by these algorithms: *dataset* and *codeblock*. In *dataset*, it should be considered that  $TC_i$  is a reference for the test case.  $P_i$  is a reference for the path built by the source code while it is interpreted. Defects detected in the SUVV are sent to  $DD_i$  which represents a list.  $IN_{ij}$ ,  $IV_{ij}$  are respectively the names and values of input variables generated for the test case ( $TC_i$ ). The same analogy is used in  $ON_{ik}$  and  $OV_{ik}$ , but for outputs in this case. And finally,  $VLN_{il}$  and  $VLV_{il}$  composes respectively the list of names and values of variables manipulated by SUVV.

$$codeblock = \{ tuple \parallel tuple = CB, PCB_i, NCB_i, CBT, CL_i \}$$

$$\begin{aligned}
CB &= \{ CB \parallel CB : \text{current code block reference} \} \\
PCB &= \{ PCB_i \parallel PCB_i : \text{previous code block list} \} \\
NCB &= \{ NCB_i \parallel NCB_i : \text{next code block list} \} \\
CBT &= \{ CBT \parallel CBT : \text{current code block type} \} \\
CL &= \{ CL_i \parallel CL_i : \text{list of code lines} \}
\end{aligned}$$

Five elements compose the *codeblock* structure.  $CB$  represents a reference for the current code block.  $PCB_i$  is a list of previous code blocks that have the current code block as destiny. The list of upcoming next code blocks which leave the current one is represented by  $NCB_i$ . There are basically two types of code blocks, code blocks which contain control structure commands, and code blocks that contain non structural commands. This information is represented by  $CBT$ . And finally,  $CL_i$  represents a list of code lines that represents the current code block.

Algorithm 4 presents a loop that performs the interpretation of each test case (line 2). The input variables of the respective test case are set in line 9, and the other variables are set as *null* (line 11). The interpretation starts by the first code block of the *main* method (lines 15 and 17), and so, this code block is read (line 18) calling Algorithm 5 recursively. The detection of defects (line 19) is done for each path resulting from test cases. And finally, the loop in line 20 and 21 set the expected output inferred from interpretation.

A subroutine called “read\_code\_block” is represented by Algorithm 5. A condition (line 3) checks if the current code block represents a control structure or not. If so, the control structure condition is checked in order to search the next code block (line 4) that may be read calling the current subroutine recursively (line 5). Otherwise, the code lines of the current code block are analyzed one by one (loop in line 7). In case of a code line that calls a method or constructor (line 9), its reference is taken (line 10) as its first code block (line 11). So, the current subroutine is called recursively with the next code block as parameter. Otherwise, the subroutine “read\_code\_line” is called (line 14).

The “read\_code\_line” subroutine is presented in Algorithm 6, where a condition checks if the code line has a value assignment (line 3), and in this case, the variable name and value assigned are taken (lines 6 and 7) and the variable list is updated (line 10). Otherwise, the code line is sent to “parse\_code\_line”, which employs several different analysis depending of the code line classification. Finally, the detection of defects at the code line level is done in line 18 and the current code line is added in the test case path in line 19.

**Input:** SUVV Model

**Output:** Expected Result for Test Case

```
1 suvv ← SUVV Model;
2 for  $TC_i \in suvv$  do
3    $VLN_{il} \leftarrow build\_variable\_list\_names(suvv, TC_i)$ ;
4    $j \leftarrow 0$ ;
5   for  $VLN_{il} \in VLN_i$  do
6     for  $IN_{ij} \in IN_i$  do
7       if  $IN_{ij} = VLN_{il}$  then
8          $IV_{ij} \leftarrow VLV_{il}$ ;
9       else
10         $IV_{ij} \leftarrow null$ ;
11      end
12    end
13  end
14   $m \leftarrow return\_method(suvv, "main")$ ;
15   $codeblock \leftarrow return\_code\_block(m, 0)$ ;
16   $read\_code\_block(suvv, codeblock, P_i)$ ;
17  /* Detection of Defects */
18   $DD_i \leftarrow check\_path\_level\_defects(P_i)$ ;
19  for  $VLN_{il} \in VLN_i$  do
20    for  $ON_{ik} \in ON_i$  do
21      if  $ON_{ik} = VLN_{il}$  then
22        /* Setting expected outputs */
23         $OV_{ik} \leftarrow VLV_{il}$ ;
24      end
25    end
26  end
27 end
```

**Algoritmo 4:** Algorithm of the source code interpretation (Part I).

**Input:** SUVV Model, Code Block

**Output:** -

```
1 subroutine read_code_block(suvv, codeblock)
2 if  $CB \neq null$  then
3   if  $CBT = "structure"$  then
4      $nextCodeBlock \leftarrow check\_control\_structure\_condition(codeblock);$ 
5      $read\_code\_block(suvv, nextCodeBlock, P_i);$ 
6   else
7     for  $codeline \leftarrow CL_i$  do
8       if  $codeline \neq null$  then
9         if  $has\_method\_call(codeline) = true$  then
10           $m \leftarrow return\_method(suvv, codeline);$ 
11           $codeblock \leftarrow return\_code\_block(m, 0);$ 
12           $read\_code\_block(suvv, codeblock, P_i);$ 
13        end
14         $read\_code\_line(suvv, codeline, P_i);$ 
15      end
16    end
17  end
18 end
```

**Algoritmo 5:** Algorithm of the source code interpretation (Part II).

**Input:** SUVV Model, Code line

**Output:** -

```
1 subroutine read_code_line(suvv, codeline)
2 if codeline  $\neq$  null then
3   if contains(codeline , " = ") = true then
4     posMid = search_character_position(codeline, " = ");
5     posEnd = search_character_position(codeline, ";");
6     varName = search_substring(codeline, 0, posMid);
7     varValue = search_substring(codeline, posMid + 1, posEnd);
8     for  $VLN_{ij} \in VLN_i$  do
9       if varName =  $VLN_{ij}$  then
10        | /* Setting variables */
11        |  $VLV_{ij} \leftarrow varValue$ ;
12        | end
13      end
14   else
15     | parse_code_line(codeline);
16   end
17   /* Detection of Defects */
18    $DD_i \leftarrow check\_line\_level\_defects(codeline)$ ;
19   add( $P_i$  , codeline);
20 end
```

**Algoritmo 6:** Algorithm of the source code interpretation (Part III).

In order to demonstrate the interpretation of a source code, consider a running example of *factorial problem* based on its tables of variables (Table 3.11) and scopes (Table 3.12). The interpreted source code forms a path, that consists of a tuple composed by the current scope (*Cs*), the code line in interpretation, input parameters (if any parameters) and elapsed variables in a specific scope, as follows:

$$path = \{ tuple \parallel tuple = Cs , Cl , Ip , Ev \}$$

$$Cs = \{ Cs_i \parallel Cs_i : current\ scope \}$$

$$Cl = \{ Cl_i \parallel Cl_i : code\ line \}$$

$$Ip = \{ Ip_i \parallel Ip_i : input\ parameters \}$$

$$Ev = \{ Ev_i \parallel Ev_i : elapsed\ variables \}$$

The interpreted source code forms a path, that consists of a tuple composed by the current scope (*Cs*), the code line, input parameters (if any) and elapsed variables in a specific scope. The demonstration is shown in Table 3.13 and considers that the input variable *number* for this test case is equal to 7. Finally, the last variable values set by interpretation are listed in Table 3.14, and the variable that was configured as SUVV output by tester (*factorial* variable) has its value stored as the expected result for this test case.

The demonstration is shown in Table 3.13 and considers that the input variable *number* for this test case is equal to 7. Note that the last variable values set by interpretation are listed in Table 3.14, and the variable that was configured as SUVV output by tester (*factorial* variable) has its value stored as the expected result for this test case.

Table 3.13 - Running example of *factorial* problem.

<b>i</b>	<b>Cs</b>	<b>Cl</b>	<b>Ip</b>	<b>Ev</b>
<b>1</b>	Factorial fat;	2	null	fat, number, varTOG_0, factorial
<b>2</b>	fat = new Factorial();	2	null	fat, number, varTOG_0, factorial
<b>3</b>	long number;	2	null	fat, number, varTOG_0, factorial
<b>4</b>	varTOG_0 = 7;	2	null	fat, number, varTOG_0 = 7, factorial
<b>5</b>	number = varTOG_0;	2	null	fat, number = 7, varTOG_0 = 7, factorial
<b>6</b>	long factorial;	2	null	fat, number = 7, varTOG_0 = 7, factorial
<b>7</b>	long fat;	1	7	n = 7, fat, varTOG_1, i, varTOG_3, varTOG_2
<b>8</b>	varTOG_1 = 1;	1	7	n = 7, fat, varTOG_1 = 1, i, varTOG_3, varTOG_2
<b>9</b>	fat = varTOG_1;	1	7	n = 7, fat = 1, varTOG_1 = 1, i, varTOG_3, varTOG_2
<b>10</b>	int i;	1	7	n = 7, fat = 1, varTOG_1 = 1, i, varTOG_3, varTOG_2
<b>11</b>	varTOG_3 = 1;	1	7	n = 7, fat = 1, varTOG_1 = 1, i, varTOG_3 = 1, varTOG_2
<b>12</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 1, varTOG_1 = 1, i = 2, varTOG_3 = 1, varTOG_2
<b>13</b>	varTOG_2 = fat * i;	1	7	n = 7, fat = 1, varTOG_1 = 1, i = 2, varTOG_3 = 1, varTOG_2 = 2
<b>14</b>	fat = varTOG_2;	1	7	n = 7, fat = 2, varTOG_1 = 1, i = 2, varTOG_3 = 1, varTOG_2 = 2
<b>15</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 2, varTOG_1 = 1, i = 3, varTOG_3 = 1, varTOG_2 = 2
<b>16</b>	varTOG_2 = fat * i;	1	7	n = 7, fat = 2, varTOG_1 = 1, i = 3, varTOG_3 = 1, varTOG_2 = 6
<b>17</b>	fat = varTOG_2;	1	7	n = 7, fat = 6, varTOG_1 = 1, i = 3, varTOG_3 = 1, varTOG_2 = 6
<b>18</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 6, varTOG_1 = 1, i = 4, varTOG_3 = 1, varTOG_2 = 6
<b>19</b>	varTOG_2 = fat * i;	1	7	n = 7, fat = 6, varTOG_1 = 1, i = 4, varTOG_3 = 1, varTOG_2 = 24

(Continue)

Table 3.13 - Continuation

<b>i</b>	<b>Cs</b>	<b>Cl</b>	<b>Ip</b>	<b>Ev</b>
<b>20</b>	fat = varTOG_2;	1	7	n = 7, fat = 24, varTOG_1 = 1, i = 4, varTOG_3 = 1, varTOG_2 = 24
<b>21</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 24, varTOG_1 = 1, i = 5, varTOG_3 = 1, varTOG_2 = 24
<b>22</b>	varTOG_2 = fat * i;	1	7	n = 7, fat = 24, varTOG_1 = 1, i = 5, varTOG_3 = 1, varTOG_2 = 120
<b>23</b>	fat = varTOG_2;	1	7	n = 7, fat = 120, varTOG_1 = 1, i = 5, varTOG_3 = 1, varTOG_2 = 120
<b>24</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 120, varTOG_1 = 1, i = 6, varTOG_3 = 1, varTOG_2 = 120
<b>25</b>	varTOG_2 = fat * i;	1	7	n = 7, fat = 120, varTOG_1 = 1, i = 6, varTOG_3 = 1, varTOG_2 = 720
<b>26</b>	fat = varTOG_2;	1	7	n = 7, fat = 720, varTOG_1 = 1, i = 6, varTOG_3 = 1, varTOG_2 = 720
<b>27</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 720, varTOG_1 = 1, i = 7, varTOG_3 = 1, varTOG_2 = 720
<b>28</b>	varTOG_2 = fat * i;	1	7	n = 7, fat = 5040, varTOG_1 = 1, i = 7, varTOG_3 = 1, varTOG_2 = 5040
<b>29</b>	fat = varTOG_2;	1	7	n = 7, fat = 5040, varTOG_1 = 1, i = 7, varTOG_3 = 1, varTOG_2 = 5040
<b>30</b>	for(i=varTOG_3;i<=n;i++){	1	7	n = 7, fat = 5040, varTOG_1 = 1, i = 8, varTOG_3 = 1, varTOG_2 = 5040
<b>31</b>	return fat;	1	7	n = 7, fat = 5040, varTOG_1 = 1, i = 8, varTOG_3 = 1, varTOG_2 = 5040
<b>32</b>	factorial = fat.factor(number);	2	null	fat, number = 7, varTOG_0 = 7, factorial = 5040
<b>33</b>	System.out.println(factorial);	2	null	fat, number = 7, varTOG_0 = 7, factorial = 5040



Table 3.14 - Last values set in *factorial problem*.

Variable	Scope	Type	Value
<b>n</b>	1	long	7
<b>fat</b>	1	long	5040
<b>varTOG_1</b>	1	-	1
<b>i</b>	1	int	8
<b>varTOG_3</b>	1	-	1
<b>varTOG_2</b>	1	-	5040
<b>fat</b>	2	Factorial	“Instance of Factorial”
<b>number</b>	2	long	7
<b>varTOG_0</b>	2	-	7
<b>factorial</b>	2	long	5040

As mentioned before, during source code interpretation, REACTOR combines static and dynamic information for detecting defects by searching for patterns that can be considered defective considering values of variables involved.

### 3.4 Static Code Analysis

In view of that inspections can detect most of software defects (SOMMERVILLE, 2010), the intention of the hybrid approach implemented in REACTOR is to perform a similar inspection automatically and with little manual labor. Thus, in this work it is used an automatic combination of static and dynamic information to detect defects during the interpretation of SUVV with test cases.

This approach was chosen since traditional static analysis algorithms can detect many software defects and extract a lot of information from the SUVV, but with limitations. For example, which code block inside a control structure is actually exercised may depend on the data that the SUVV is handling. This information can only be addressed based on dynamic information, which consists in monitoring variable values and instrumenting the source code to produce information regarding exercised paths (AGGARWAL; JALOTE, 2006).

Therefore, REACTOR can detect defects based on particular standards found in the source code which are exercised for each test case, and it may be checked by tester to decide whether or not the code should be corrected. Certain defects can only be

addressed by combining static analysis with dynamic information. In REACTOR, it was defined 47 types of defects which are divided into six basic classes that are described below.

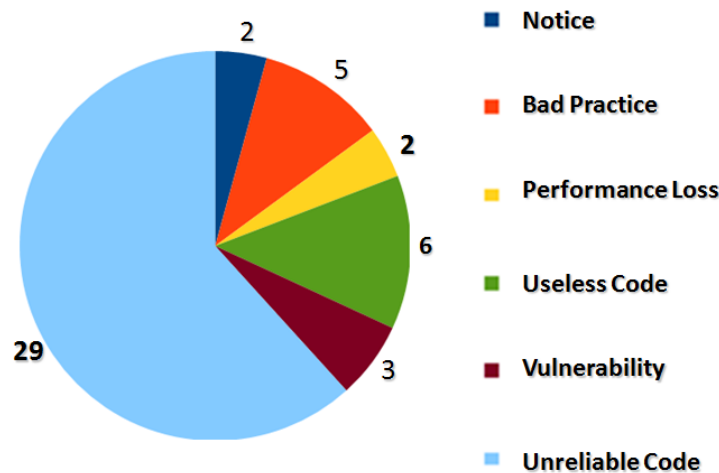
- **Notice:** warning messages about details that the tester should know, but that are not probably the cause of a defect directly, such as an access of an external file or database.
- **Bad Practice:** warnings about questionable coding practices that may not belong to usual conventions or programming standards.
- **Performance Loss:** parts of the source code that can cause loss of performance, as very complex control structures or print commands inside much repeated loops.
- **Useless Code:** classes, methods, constructors, variables or code blocks that are not exercised by test cases.
- **Vulnerability:** programming faults that are not explored by test cases, but can be exploited by malicious attacks, such as public constructors or public variables.
- **Unreliable Code:** parts of the source code that can cause an error or exception depending on input data loaded into its variables, such as a division by zero or numeric overflows.

The detection of static defects is done in three stages: static and dynamic analysis of an exercised path for each test case, static and dynamic analysis for each code line exercised also during the interpretation, and static analysis of the entire SUVV model (performed just once).

Figure 3.12 illustrates the proportion of the number of defects that can be detected by each class implemented in REACTOR. Note that REACTOR implements most of them related to unreliable code (29), followed by useless code (6), and these both categories make use of dynamic information in order to detect defects. Other static class that use dynamic information is performance loss. Therefore, the most of defects that can be detected by REACTOR are not addressed by other tools which performs only the static analysis.

Algorithms 7 to 16 present a summarized version for the detection of some types of defects by REACTOR.

Figure 3.12 - REACTOR: distribution of the 47 types of static defects into classes.



Algorithms 7 and 8 detects notices related to variables which are set with values really close to its limits, and source code that depends on accessing external files, respectively. Algorithms 9 and 10 detects, respectively, the inadvisable use of constant numbers, and the wrong naming convention of classes, methods and variables. Algorithm 11 detects collapsed “ifs” that can be merged in order to increase the executing performance, and Algorithm 12 detects the use of print commands. Algorithm 13 detects classes, methods, constructors or even code blocks that were not exercised by test cases. Algorithm 14 detects the use of public constructors or attributes. And finally, Algorithms 15 and 16 present a the detection of some types of unreliable code defects. Both defects are classified in REACTOR as unreliable code, and other static analysis tools are no able to detect such defects. Algorithm 15 detects division by zero, that is a common defect in computation, and Algorithm 16 detects numeric overflows. These algorithms makes use of the same *suvv* tuple described in Section 3.2.4 and of the *testing\_step* tuple, described as follows:

$$testing\_step = \{ tuple \parallel tuple = codeline , VLN , VLV , VLT \}$$

$$codeline = \{ codeline \parallel codeline : current\ code\ line \}$$

$$VLN = \{ VLN_i \parallel VLN_i : variable\ list\ names \}$$

$$VLV = \{ VLV_i \parallel VLV_i : variable\ list\ values \}$$

$$VLT = \{ VLT_i \parallel VLT_i : variable\ list\ types \}$$

In these previous algorithms, it should consider that *testing\_step* is a tuple which is composed of four elements. *codeline* is a reference for the current code line in analysis.  $VLN_i$  is a reference for the list of variable names during the interpretation.  $VLV_i$  and  $VLT_i$  are, respectively, the list of values and types that corresponds to variables in  $VLN_i$ .

In Algorithm 7, every code line that is classified as an numeric assignment (line 4) searches for the variable type for which the value will be assigned (lines 5 to 10). So, the upper and lower limits of such type is computed (lines 11 and 12) and compared with the value that supposedly should be assigned (line 13). If the value is greater than the upper limit (line 14) or lower than the lower limit (line 17), it is reported a notice that it is close to variable limit. And Algorithm 8 has a loop (line 1) which examine the list of variables searching for variables of the types *BufferedReader* or *FileReader* (line 2). If there is any variable of this type, a file dependence is noticed (line 4).

**Input:** Code Line and Dynamic Information

**Output:** Notice Code Defect

```

1 posMid ← search_character_position(codeline , " = ");
2 posEnd ← search_character_position(codeline , ";" );
3 varName ← search_substring(codeline , 0 , posMid);
4 if codeline is classified as numeric assignment then
5   for  $VLN_i \in VLN$  do
6     if varName =  $VLN_i$  then
7       /* Getting variable type */
8       varType ←  $VLT_i$ ;
9     end
10  end
11  upperLimit ← upper_type_limit(varType);
12  lowerLimit ← lower_type_limit(varType);
13  varValue ← search_substring(codeline , posMid + 1 , posEnd);
14  if varValue ≥ ( upperLimit * 0.9 ) then
15    report_defect_detected("upper limit dangerously close");
16  end
17  if varValue ≤ ( lowerLimit * 0.9 ) then
18    report_defect_detected("lower limit dangerously close");
19  end
20 end

```

**Algoritmo 7:** Algorithm to detect notices (close to variable limit).

**Input:** Code Line ad Dynamic Information

**Output:** Notice Code Defect

```
1 for  $VLN_i \in VLN$  do
2   if  $VLT_i = \text{"BufferedReader"} \vee VLT_i = \text{"FileReader"}$  then
3     if  $VLV_i \neq \text{"null"}$  then
4        $report\_defect\_detected(\text{"file dependence"})$ ;
5     end
6   end
7 end
```

**Algoritmo 8:** Algorithm to detect notices (file dependence).

Algorithm 9 presents three loops which read each code line (line 6) of each method (line 3) within SUVV classes (line 2). If the code line is not in *main* method (line 7) and if it is a numeric assignment (line 8) or a hex assignment (line 11), it is reported a bad practice of using magic constants (lines 9 and 12). And Algorithm 10 performs three loops (line 2, 6 and 12) which examines the names of classes, methods and variables with determined *regex* (line 3, 7 and 13) in order to find names that does not follow the naming convention.

**Input:** Code Line ad Dynamic Information

**Output:** Bad Practice Code Defect

```
1  $suvv \leftarrow SUVV\ Model$ ;
2 for  $C_i \in suvv$  do
3   for  $M_i \in C_i$  do
4      $j \leftarrow 0$ ;
5      $k \leftarrow 0$ ;
6     for  $CL_{ij}^c \in M_i$  do
7       if  $M_i$  is not the "main" method then
8         if  $check\_classification(CL_{ij}^c) = \text{"assignment numeric"}$  then
9            $report\_defect\_detected(\text{"magic number"})$ ;
10        end
11        if  $check\_classification(CL_{ij}^c) = \text{"assignment hexadecimal"}$  then
12           $report\_defect\_detected(\text{"magic hexadecimal"})$ ;
13        end
14      end
15    end
16  end
17 end
```

**Algoritmo 9:** Algorithm to detect bad practice (magic number).

**Input:** Code Line ad Dynamic Information

**Output:** Bad Practice Code Defect

```
1  $suvv \leftarrow SUVV$  Model;
2 for  $C_i \in suvv$  do
3   | if  $C_i$  does not matches with class name "regex" then
4   |   |  $report\_defect\_detected("Naming\ convention\ of\ class");$ 
5   | end
6   | for  $M_i \in C_i$  do
7   |   | if  $M_i$  does not matches with method name "regex" then
8   |   |   |  $report\_defect\_detected("Naming\ convention\ of\ method");$ 
9   |   |   | end
10  |   | end
11 end
12 for  $VLN_i \in VLN$  do
13  | if  $VLN_i$  does not matches with variable name "regex" then
14  |   |  $report\_defect\_detected("Naming\ convention\ of\ variables");$ 
15  |   | end
16 end
```

**Algoritmo 10:** Algorithm to detect bad practice (naming convention).

Algorithm 11 presents a set of loops which examines inside the source code (line 10) exercised by test cases (line 7) if there is two “ifs” (lines 11 and 14) declared in sequence, and so, that can be merged in only one code line. And Algorithm 12, each code line is examined in order to find print command lines (line 7).

**Input:** Code Line and Dynamic Information

**Output:** Performance Loss Code Defect

```
1 suvv ← SUVV Model;
2 for  $C_i \in suvv$  do
3   for  $M_i \in C_i$  do
4     flag1 ← false;
5     flag2 ← false;
6     flag3 ← false;
7     if is_used( $M_i$ ) = true then
8        $j \leftarrow 0$ ;
9        $k \leftarrow 0$ ;
10      for  $CL_{ij}^c \in M_i$  do
11        if check_classification( $CL_{ij}^c$ ) = "if opening" then
12          flag1 ← true;
13          for  $CL_{ik}^c \in M_i$  do
14            if check_classification( $CL_{ik}^c$ ) = "if opening" then
15              flag2 ← true;
16              flag3 ← true;
17              for  $x = j + 1$  (to)  $x < k$  do
18                 $x = x + 1$ ;
19                 $cl = CL_{ix}^c$ ;
20                if cl does not starts with temporary variable then
21                  | flag3 = false;
22                end
23              end
24            end
25          end
26          res = false;
27          if flag1 = true  $\wedge$  flag2 = true  $\wedge$  flag3 = true then
28            | res = true;
29          end
30        end
31      end
32    end
33    if res = true then
34      | report_defect_detected("collapsed ifs can be merged");
35    end
36  end
37 end
```

**Algoritmo 11:** Algorithm to detect performance loss (collapsed “ifs”).

**Input:** Code Line and Dynamic Information  
**Output:** Performance Loss Code Defect

```

1  $suvv \leftarrow SUVV \text{ Model};$ 
2 for  $C_i \in suvv$  do
3   for  $M_i \in C_i$  do
4     if  $is\_used(M_i) = true$  then
5        $j \leftarrow 0;$ 
6       for  $CL_{ij}^c \in M_i$  do
7         if  $check\_classification(CL_{ij}^c) = "print \text{ line}"$  then
8            $report\_defect\_detected("print \text{ command as logger}");$ 
9         end
10      end
11    end
12  end
13 end

```

**Algorithm 12:** Algorithm to detect performance loss (print command as logger).

Algorithm 13 uses an artifact implemented in SUVV model that is the *isUsed* attribute (shown in Section 3.2.6). So, useless code defects are found by checking this attribute within classes, methods/constructors and code blocks after the interpretation of test cases. And Algorithm 14 uses another artifact provided by the SUVV model, that is the *visibility* attribute. So, it is possible to find constructors and attributes set with public visibility based on few loops.

It is interesting to cite that Algorithms 11, 12, 13 and 14 only detect defects in classes/methods/constructors that were exercised by test cases (note that there is a *is\_used* function within “ifs” which consults the *isUsed* attribute). So, some types of defects are not purposely detected within useless code, which assumes that it can be disposed of by the developer. This feature contributes to the reduction of false alarms.



**Input:** Code Line and Dynamic Information

**Output:** Useless Code Defect

```
1  $suwv \leftarrow SUVV$  Model;  
2 for  $C_i \in suwv$  do  
3   if  $is\_used(C_i) = false$  then  
4      $report\_defect\_detected("useless\ class");$   
5   else  
6     for  $M_i \in C_i$  do  
7       if  $is\_used(M_i) = false$  then  
8          $report\_defect\_detected("useless\ method\ or\ constructor");$   
9       end  
10      for  $codeblock \in M_i$  do  
11        if  $is\_used(codeblock) = false$  then  
12           $report\_defect\_detected("useless\ code\ block");$   
13        end  
14      end  
15    end  
16  end  
17 end
```

**Algoritmo 13:** Algorithm to detect useless code (class, method/constructor and code block).

**Input:** Code Line and Dynamic Information  
**Output:** Vulnerability Defect

```
1 suvv ← SUVV Model;  
2 for  $C_i \in suvv$  do  
3   if is_used( $C_i$ ) = true then  
4     for  $M_i \in C_i$  do  
5       if is_used( $M_i$ ) = true then  
6         if check_visibility( $C_i$ ) = "public" then  
7           if check_name( $C_i$ ) = check_name( $M_i$ ) then  
8             report_defect_detected("public visibility constructor");  
9           end  
10        end  
11       end  
12     end  
13   else  
14 end  
15 for  $VLN_i \in VLN$  do  
16   if check_visibility( $VLN_i$ ) = "public" then  
17     if check_scope( $VLN_i$ ) = "attribute" then  
18       report_defect_detected("public visibility attribute");  
19     end  
20   end  
21 end
```

**Algoritmo 14:** Algorithm to detect vulnerability (public constructors or attributes).

In Algorithm 15, every code line that is classified as an division operation (line 4 and 5) performs a checking of divider value which is accessed from the variable lists  $VLN_i$ ,  $VLV_i$  and  $VLT_i$  (lines 9 to 14). If the divider value is equal to zero (line 15) it is reported a division by zero defect (line 16). And in Algorithm 16, every code line that is classified as an numeric assignment searches for the variable type for which the value will be assigned (lines 5 to 10). So, the upper and lower limits of such type is computed (lines 11 and 12) and compared with the value that supposedly should be assigned (line 13). If the value is greater that the upper limit (line 14) or lower that the lower limit (line 17), it is reported a numeric overflow defect.

**Input:** Code Line ad Dynamic Information

**Output:** Unreliable Code Defect

```

1 posMid ← search_character_position(codeline , " = ");
2 posEnd ← search_character_position(codeline , ";"");
3 varName ← search_substring(codeline , 0 , posMid);
4 if codeline is classified as operation between variables then
5   if operator is division then
6     posOp ← search_character_position(codeline , operator);
7     varNameR ← search_substring(codeline , posOp + 1 , posEnd);
8     varNameL ← search_substring(codeline , posMid + 1 , posOp);
9     for  $VLN_i \in VLN$  do
10      if varNameR =  $VLN_i$  then
11        /* Getting variable value */
12        varValueR ←  $VLV_i$ ;
13      end
14    end
15    if varValueR = 0 then
16      report_defect_detected("division by zero");
17    end
18  end
19 end

```

**Algoritmo 15:** Algorithm to detect unreliable code (division by zero).

**Input:** Code Line and Dynamic Information

**Output:** Unreliable Code Defect

```
1 posMid ← search_character_position(codeline , "=");
2 posEnd ← search_character_position(codeline , ";");
3 varName ← search_substring(codeline , 0 , posMid);
4 if codeline is classified as numeric assignment then
5   for  $VLN_i \in VLN$  do
6     if varName =  $VLN_i$  then
7       /* Getting variable type */
8       varType ←  $VLT_i$ ;
9     end
10  end
11  upperLimit ← upper_type_limit(varType);
12  lowerLimit ← lower_type_limit(varType);
13  varValue ← search_substring(codeline , posMid + 1 , posEnd);
14  if varValue ≥ upperLimit then
15    | report_defect_detected("upper limit reached");
16  end
17  if varValue ≤ lowerLimit then
18    | report_defect_detected("lower limit reached");
19  end
20 end
```

**Algoritmo 16:** Algorithm to detect unreliable code (numeric overflow).

With respect to the *factorial problem*, five defects were detected by REACTOR as presented in Figure 3.13. There are two bad practice issues, one performance loss code part and two unreliable code lines.

Figure 3.13 - Defects detected in *factorial problem*.

```
1 package iut.TestFactorial; ← Bad Practice
2
3 public class Factorial {
4
5     public long factor(long n) {
6         long fat = 1; ← Bad Practice
7         for (int i = 1; i <= n; i++) { ← Unreliable Code
8             fat = fat * i; ← Unreliable Code
9         }
10        return fat;
11    }
12
13    public static void main(String[] args) {
14        Factorial fat = new Factorial();
15        long number = 7;
16        long factorial = fat.factor(number);
17        System.out.println(factorial); ← Performance Loss
18    }
19 }
```

The first bad practice issue is related to a package name which should comply with a naming convention, and the second one is related to the use of what static analysis tools call “magic number” or “magic constants”, and the use of constants spread in the source code must be discouraged.

A performance loss was detected in a code line that performs a print command. This is considered a performance loss defect by static analysis tools since the print command consumes much more execution time than other loggers that could be used. The detection of this defect is specially useful when print commands are within very repeated loops.

Finally, unreliable code lines were detected in two code lines that can potentially raise an exception. Note that these source code lines can cause a failure if they are executed with a very high value set in *number* variable. This type of defect can only be detected by using static analysis combined with dynamic information, therefore it can not be detected by tools that perform only static analysis.

### 3.5 Oracle Procedure

REACTOR determines verdicts of the test cases by comparing (estimated) expected with actual results. Expected results are inferred according to what was discussed in Section 3.3; on the other hand, actual results may be obtained by executing SUVV with the same test cases that inferred expected results. For this purpose, REACTOR instruments SUVV automatically so that it can execute with the same set of test cases.

This instrumented SUVV implements the original SUVV by changing it so that it works as a *driver*. So, it changes only the class that contains a *main* method and creating a changed copy of it. The copy of SUVV changes the acquisition of the input in order to get test case inputs, and by redirecting the output data to set the test case results.

Once test oracle information (expected results) is based only on the source, it does not require an explicit representation of the requirements. Hence, the oracle information sometimes will not be the true expected result that it would obtain if our oracle approach demanded the documentation to be considered too, so it was preferred call it “estimated oracle information”. Therefore, although it is generated an “expected result”, this outcome is not really too relevant to our approach because our oracle is much more like an organic oracle where, combined with our automated test case generation strategy, it is aimed to address one specific type of testing defect: exception handling.

An oracle procedure performs an automatic comparison between expected results from the generated oracle and actual results from SUVV execution. The overall result from REACTOR is a set of test cases with their respective verdicts, and a list of testing defects detected within the SUVV. REACTOR considers there are three possible verdicts:

- **Pass:** when a valid estimated result from a test case is exactly the same as actual result.
- **No pass:** when a valid estimated result from a test case is different from actual result.
- **Inconclusive:** when the estimated result and actual result returns invalid results, which can not be compared.

Several types of testing defects can be detected, and as explained in Section 2.1, a defect can or can not become a failure. In REACTOR, beyond the detection of defects, failures can also be found when expected results are not the same as actual results obtained via testing. So, in REACTOR, it can be said that testing is a complementary technique used to prove or disprove the existence of a defect that becomes a real failure, and thus reduce the probability of detecting false positives. The set of information used by oracle procedure is defined follows:

$$oracle = \{ tuple \parallel tuple = TC , VN , ER , AR , V \}$$

$$TC = \{ TC_i \parallel TC_i : suvv \text{ test case reference} \}$$

$$VN = \{ VN_i \parallel VN_i : output \text{ variable name} \}$$

$$ER = \{ ER_{ij} \parallel ER_{ij} : expected \text{ result} \}$$

$$AR = \{ AR_{ij} \parallel AR_{ij} : actual \text{ result} \}$$

$$V = \{ V_i \parallel V_i : test \text{ case verdict} \}$$

Algorithm 17 shows how the test oracle procedure operates, and for this algorithm, it may be considered that *oracle* is a tuple which is composed by five elements.  $TC_i$  is a reference for the test case.  $VN_{ij}$  composes the list of output variable names.  $ER_{ij}$  and  $AR_{ij}$  are respectively the expected and actual results for the output variable named by  $VN_{ij}$ . And finally,  $V_i$  is the final verdict inferred by the procedure component.

Note that Algorithm 17 implements a loop that performs the procedure for each test case (line 2). The verdict is set as “pass” (line 3). If expected and actual results are valid (line 6) they may be compared (line 8), and if they are not equal verdict is set as “no pass” (line 9). If one of them (or both) are invalid, verdict is set as “inconclusive” (line 12). Finally, the *oracle* is set with the final verdict (line 16), and once all test cases were performed, the HTML table is saved in file system line 19).

**Input:** Test Oracle Information

**Output:** Test Oracle Verdicts

```
1 fw ← File Writer;  
2 for  $TC_i \in \text{oracle}$  do  
3   verdict ← "pass";  
4   for  $VN_{ij} \leftarrow VN_i$  do  
5     /* both valid results ? */  
6     if  $is\_valid(ER_{ij}) \wedge is\_valid(ARN_{ij})$  then  
7       /* comparing results */  
8       if  $ER_{ij} \neq ARN_{ij}$  then  
9         | verdict ← "nopass";  
10        end  
11       else  
12         | verdict ← "inconclusive";  
13        end  
14      end  
15      /* setting verdict */  
16       $V_i \leftarrow verdict$ ;  
17 end  
18 /* saving HTML table */  
19 save_oracle_table(fw , oracle);
```

**Algoritmo 17:** Algorithm of the test oracle procedure.



### 3.6 Final Remarks

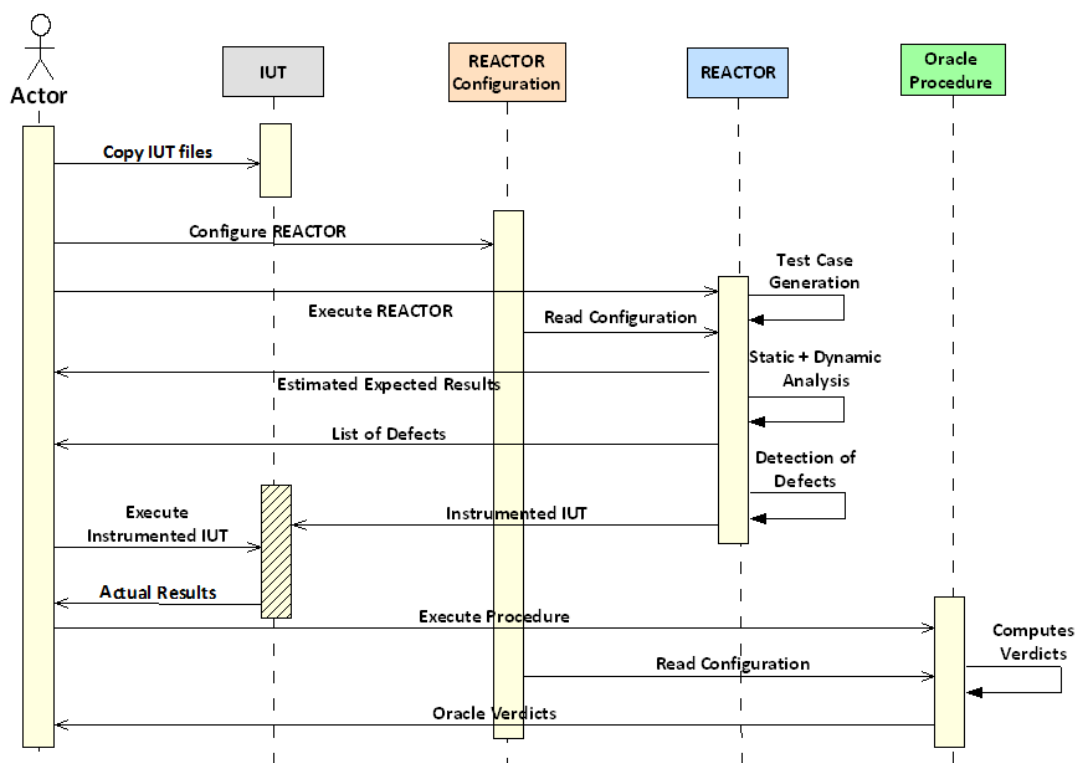
Published literature proposed solutions for detecting defects via static analysis, however, there are several types of defects that can not be revealed by presented approaches. Thus, this thesis proposed the combined use of static and dynamic information in a way that results obtained dynamically are used to reinforce possible defects found in static analysis. REACTOR performs a similar solution by an automated approach for detecting defects by generating test oracle information and procedure. REACTOR tool works by analyzing SUVV by reverse engineering and building an SUVV model based on class models. Based on a set of test cases, an estimation of expected results are inferred based on the interpretation of source code, and verdicts are determined by comparing expected results with actual results obtained by SUVV execution.



## 4 IMPLEMENTATION OF REACTOR

This chapter presents development aspects of the REACTOR method by describing its software architecture that implements the main concepts presented in Chapter 3. It will also address operation of the tool based on the sequence diagram in Figure 4.1, which illustrates the interactions required by the tester beyond the collaboration among REACTOR's objects based on a time sequence.

Figure 4.1 - REACTOR's sequence diagram.

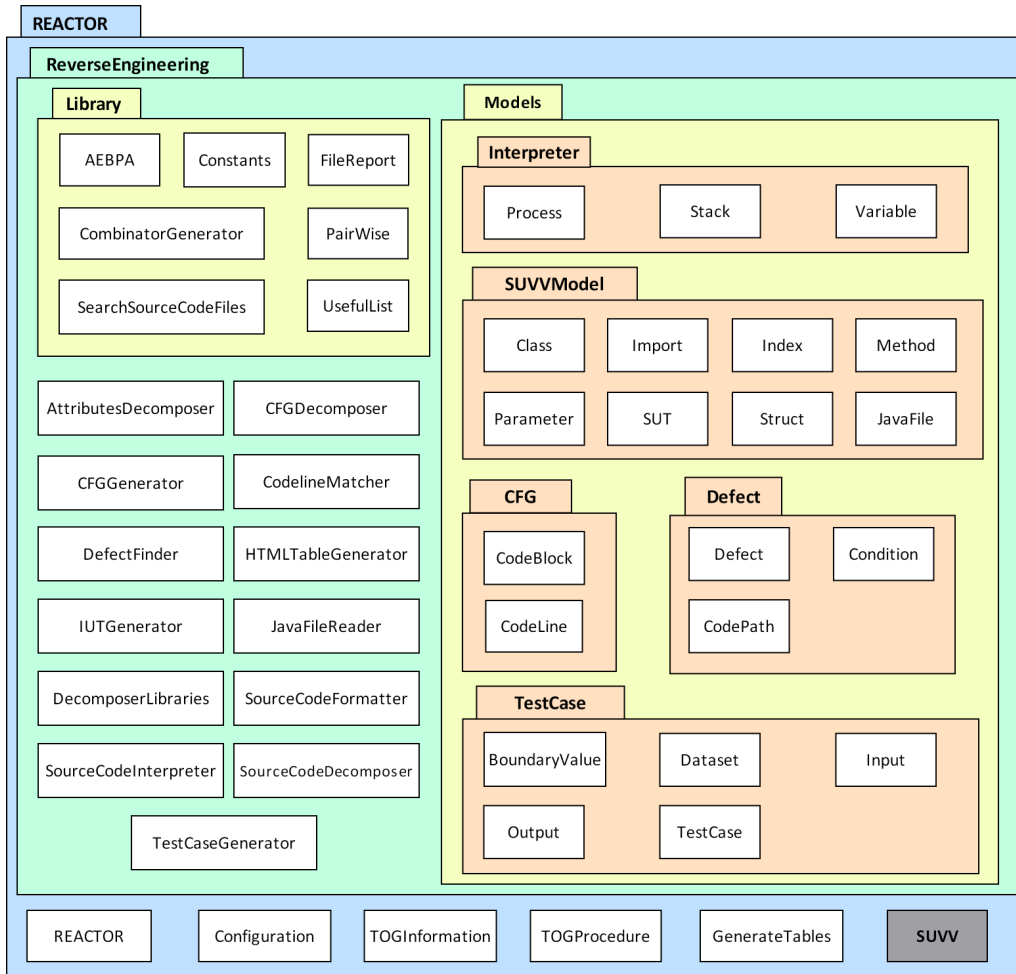


### 4.1 REACTOR's Architecture

The development of REACTOR made use of Java JDK (Java Development Kit) 1.7.0, and does not make access to any database management system. Figure 4.2 shows the package diagram which groups all classes used by the tool.

REACTOR package incorporates several sub-packages arranged according to specific scopes. The package "*Library*" contains some useful classes which are used by the tool, as the list of reserved words of Java language, constants, the combiner of

Figure 4.2 - REACTOR's package diagram.



inputs for the automatic generation of test cases, the AEBPA for test case generation and pairwise algorithm.

“*Models*” contains several sub-packages with the prospect of reverse engineering. “*Interpreter*” is composed by three classes which are used to interpret the classified source code. “*SUVVModel*” is composed by eight classes used to represent the SUVV model. The structure of the source code files in blocks is built by two classes in “*CFG*” sub-package. The detection of defects uses three classes in “*Defect*” sub-package and the test case suite is stored by five classes in “*TestCase*” sub-package.

“*ReverseEngineering*” package contains the sub-packages “*Library*” and “*Models*”, which as mentioned, also has its own sub-packages. Beyond these two sub-packages, “*ReverseEngineering*” is composed also by thirteen classes required for reverse engi-

neering and static analysis, file reader, such as test case generation, decomposition, *regex* matcher, interpretation, instrumentation and detection of defects.

The package “*REACTOR*” comprises the complete tool which is executed by classes “*REACTOR*” and “*TOPcedure*”. The REACTOR’s configuration must be done in “*Configuration*” and the instrumented SUVV is represented by “*SUVV*”.

## 4.2 Configuration

The first steps of REACTOR once it is executed with a preconfigured SUVV, are by searching source code files within SUVV directory. As mentioned before, static analysis can only be done if source code is available, so it is a fundamental step.

The preconfiguration of SUVV’s directory, as its inputs and outputs, must be coded in a configuration class within the tool. This class contains some attributes that can be set by tester according to the SUVV. Figure 4.3 presents the configuration class of factorial problem, and Figure 4.4 presents the configuration class of triangle classification. The attributes that must be coded by tester are:

- *SUVV\_inputs*: represents all inputs that must be used to generate test cases. It must be coded as an array of Strings, and each string must be in format “type variable-name”. Note that in Figure 4.3 there is only one input variable, and in Figure 4.4 there are tree of them.
- *SUVV\_outputs*: represents all outputs that must be considered estimated expected results to compute. It must be coded by using the same format as inputs, and it must be declared in *main* method of SUVV.
- *SUVV\_input\_values*: if this it set as *null* (as shown in Figure 4.3), input values are automatically generated by AEBPA. Otherwise, tester can code particular values of interest, as shown in Figure 4.4.
- *SUVV\_JAVA\_FILES*: this is the directory path that contains all source code files of SUVV.
- *WORK\_FOLDER*: during its execution, REACTOR creates a temporary directory with several text files. This directory is created within this pre-configured folder.
- *SUVV\_ID*: this attribute must be set with a short string that identifies the SUVV. The temporary directory is created with this name.

- *PAIRWISE*: it is a boolean variable that indicates if pairwise is active to reduce the set of test cases for SUVV with three or more inputs.

Figure 4.3 - Configuration class of *factorial problem*.

```

package reactor;

import reactor.Library.Constants;

public class Config {

    public String[] IUT_inputs = {"long number"};
    public String[] IUT_outputs = {"long factorial"};
    public String[][] IUT_input_values = null;
    public String IUT_JAVA_FILES = "C:\\Java\\REACTOR\\src\\iut\\TestFactorial\\";
    public String WORK_FOLDER = "C:\\Java\\";
    public String IUT_ID = "Factorial";
    public boolean PAIRWISE = true;

}

```

Figure 4.4 - Configuration class of *triangle classification*.

```

package reactor;

import reactor.Library.Constants;

public class Config {

    public String[] IUT_inputs = {"double input_a", "double input_b", "double input_c"};
    public String[] IUT_outputs = {"String type"};
    public String[][] IUT_input_values = {{ "-1", "0", "1", {"0", "1", "2"}, {"1", "2", "3"} };
    public String IUT_JAVA_FILES = "C:\\Java\\REACTOR\\src\\iut\\TestTriangle\\";
    public String WORK_FOLDER = "C:\\Java\\";
    public String IUT_ID = "Triangle";
    public boolean PAIRWISE = true;

}

```

## 4.3 REACTOR's Execution

### 4.3.1 Automatic Test Case Generation

Next, REACTOR generates test cases automatically based on information about input and output variables that the testing professional may provide in a configura-

tion class. These are automatically planned according to the variable types involved, being tested for upper and lower limits, a fixed positive and negative value, a random positive and negative value, and zero.

The Tables 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 and 4.8 present a list of values that are used in order to generate test cases by the AEBPA approach (Section 3.1.1). It is important to mention that the approach used in this work can generate test cases just based on variables of primitive types. In addition, since all possible combinations of inputs are generated exhaustively, the combination can result in a huge amount of generated test cases. In order to deal with this situation, a pairwise algorithm (Section 3.1.2) was implemented to filter and reduce the set of test cases when SUVV has more than two input variables.

Table 4.1 - Suggested limits for testing boolean variables (1 bit).

Lower limit	“false”
Upper limit	“true”

Table 4.2 - Suggested limits for testing char variables (2 bytes/16 bits).

Lower limit	0
Fixed positive value	n
Random positive value	?
Upper limit	65535

Table 4.3 - Suggested limits for testing byte variables (1 byte/8 bits).

Lower limit	-128
Random negative value	-?
Fixed negative value	-n
Zero	0
Fixed positive value	n
Random positive value	?
Upper limit	127

Table 4.4 - Suggested limits for testing short variables (2 bytes/16 bits).

Lower limit	-32.768
Random negative value	-?
Fixed negative value	-n
Zero	0
Fixed positive value	n
Random positive value	?
Upper limit	32.767

Table 4.5 - Suggested limits for testing integer variables (4 bytes/32 bits).

Lower limit	-2.147.483.648
Random negative value	-?
Fixed negative value	-n
Zero	0
Fixed positive value	n
Random positive value	?
Upper limit	2.147.483.647



Table 4.6 - Suggested limits for testing long variables (8 bytes/64 bits).

Lower limit	-9.223.372.036.854.775.808
Random negative value	-?
Fixed negative value	-n
Zero	0
Fixed positive value	n
Random positive value	?
Upper limit	+9.223.372.036.854.775.807

Table 4.7 - Suggested limits for testing float variables (4 bytes/32 bits).

Lower limit	1.40129846432481707e-45
Random negative value	-?
Fixed negative value	-n
Zero	0
Fixed positive value	n
Random positive value	?
Upper limit	3.40282346638528860e+38

Table 4.8 - Suggested limits for testing double variables (8 bytes/64 bits).

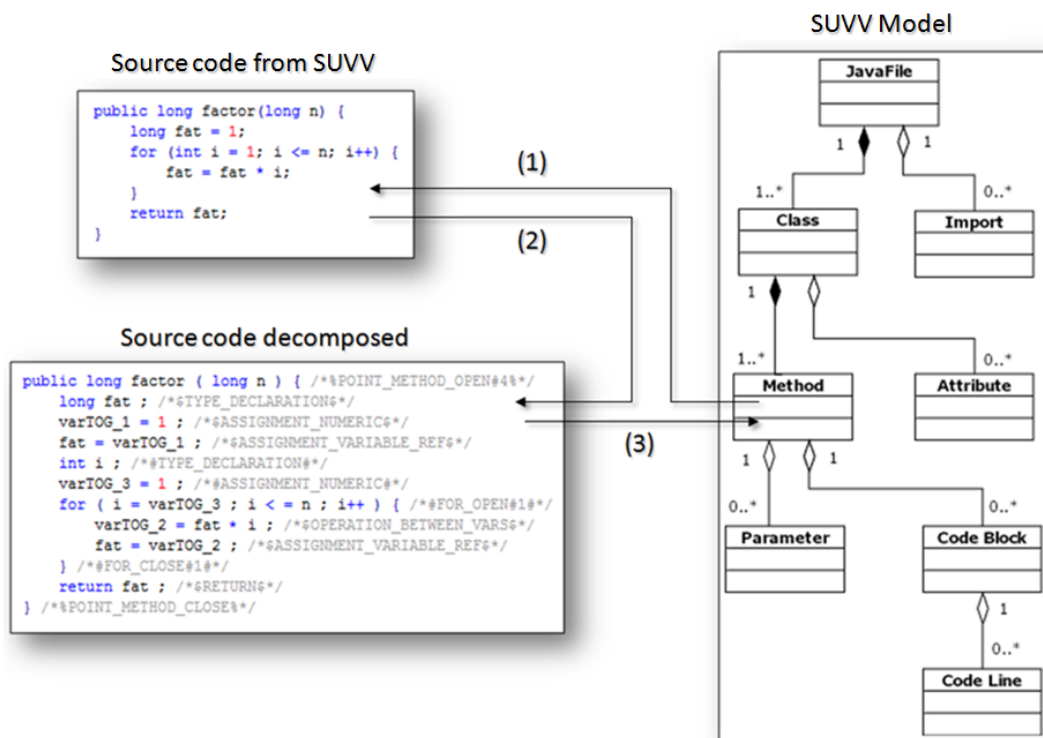
Lower limit	4.94065645841246544e-324d
Random negative value	-?
Fixed negative value	-n
Zero	0
Fixed positive value	n
Random positive value	?
Upper limit	1.79769313486231570e+308d

### 4.3.2 Automatic Reverse Engineering of Static Source Code

Once the source code is available based on SUVV analysis, it must be properly instrumented and prepared to be analyzed by *regex* (Section 3.2.1). This preparation basically consists in formatting to check the spacing and line breaks, and insert standardized markings to set locations of opening and closing scopes in classes, methods, libraries, and packages. Only then, SUVV is analyzed and recognized by predetermined expressions implemented using a *regex* language supported by libraries in Java. Finally, the structural model of the SUVV (Figure 3.7) is built in REACTOR totally by reverse engineering of Java software.

REACTOR reads each code line for each Method instance in SUVV model, so it decomposes, classifies (according to the Table A.3), and overwrites the old code lines in the model as showed in Figure 4.5.

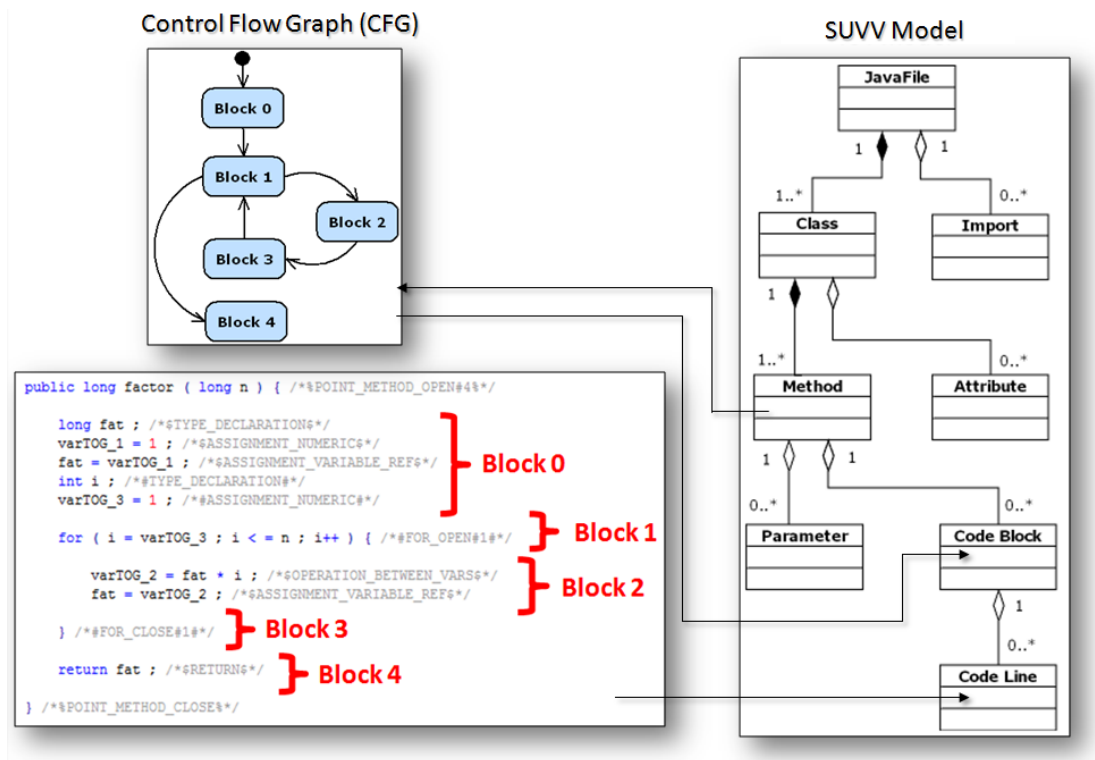
Figure 4.5 - Source code decomposition and classification.



The reading of each decomposed and classified code line, that if representing a control structure, this structure is built in a set of linked code blocks similar to a

CFG (Section 3.2.6) showed in Figure 4.6.

Figure 4.6 - Conversion of source code into CFG.



Once the model is ready with SUVV grouped in code blocks, the source code is interpreted feeding a list of variables that was set by collecting all variables of SUVV. It includes, of course, temporary variables created by the decomposition of source code. The standard used to create temporary variables was the string “varTOG\_” followed by an integer number that is automatically incremented, thus, it is unlikely that the SUVV has a coded variable with the same name causing a problem. Figure 4.7 shows *factorial problem* source code with temporary variables highlighted.

At the end of the test case, the output variables specified by the tester have their values stored as estimated expected results and, thus, this step occurs again for next test cases generated. Once the set of test cases were interpreted, a text file with inputs and estimated expected outputs is saved. This text file provides input data for a modified version of SUVV that may be executed with the same test cases, and its contents generated for *factorial problem* are shown in Figure 4.8.

Figure 4.7 - Temporary variables created by REACTOR in *factorial* problem.

```

1 package iut.TestFactorial ; /*%POINT_PACKAGE%*/
2
3 public class Factorial { /*%POINT_CLASS_OPEN#2%*/
4
5     public long factor ( long n ) { /*%POINT_METHOD_OPEN#4%*/
6         long fat ; /*%TYPE_DECLARATION%*/
7         varTOG_1 = 1 ; /*%ASSIGNMENT_NUMERIC%*/
8         fat = varTOG_1 ; /*%ASSIGNMENT_VARIABLE_REF%*/
9         int i ; /*%TYPE_DECLARATION#1%*/
10        varTOG_3 = 1 ; /*%ASSIGNMENT_NUMERIC#1%*/
11        for ( i = varTOG_3 ; i <= n ; i++ ) { /*%FOR_OPEN#1%*/
12            varTOG_2 = fat * i ; /*%OPERATION_BETWEEN_VARS%*/
13            fat = varTOG_2 ; /*%ASSIGNMENT_VARIABLE_REF%*/
14        } /*%FOR_CLOSE#1%*/
15        return fat ; /*%RETURN%*/
16    } /*%POINT_METHOD_CLOSE%*/
17
18    public static void main ( String [ ] args ) { /*%POINT_MAIN_OPEN%*/
19        Factorial fat ; /*%TYPE_DECLARATION%*/
20        fat = new Factorial () ; /*%ASSIGNMENT_NEW_OBJECT%*/
21        long number ; /*%TYPE_DECLARATION%*/
22        varTOG_0 = 7 ; /*%ASSIGNMENT_NUMERIC%*/
23        number = varTOG_0 ; /*%ASSIGNMENT_VARIABLE_REF%*/
24        long factorial ; /*%TYPE_DECLARATION%*/
25        factorial = fat.factor ( number ) ; /*%FUNCTION_RETURN_FROM_REF%*/
26        System.out.println ( factorial ) ; /*%PRINT%*/
27    } /*%POINT_MAIN_CLOSE%*/
28 } /*%POINT_CLASS_CLOSE%*/


```

### 4.3.3 Automatic Detection of Defects

During and after the interpretation of test cases, the detector of defects evaluates the behavior of SUVV (testing) stimulated by source code (static analysis) operations, and apply several techniques to search six classes of defects (as is detailed in Section 3.4). The defect is detected in three stages:

- *1st stage*: this is the code line analysis. There are detected defects during the interpretation of each singly code line, based on static and dynamic information. It is done once for each code line interpreted.
- *2nd stage*: this is the source code path analysis. There are detected defects by analyzing the path traversed by each test case in source code. It is performed once for each test case.
- *3rd stage*: this is the SUVV model analysis. There are detected defects by

Figure 4.8 - Text file with input and output data of *factorial* problem.



```
Listner - [C:\Java\IUT_Factorial_OKI... 81 %
File Edit Options Help
INPUT_DESCRIPTION
number # long

OUTPUT_DESCRIPTION
factorial @ long

TESTCASE_ID : 0
INPUT_LIST : 0
number : -94
OUTPUT_LIST : 0
factorial : 1
EXECUTE_TESTCASE

TESTCASE_ID : 1
INPUT_LIST : 1
number : 0
OUTPUT_LIST : 1
factorial : 1
EXECUTE_TESTCASE

TESTCASE_ID : 2
INPUT_LIST : 2
number : 49
OUTPUT_LIST : 2
factorial : 0
EXECUTE_TESTCASE

TESTCASE_ID : 3
INPUT_LIST : 3
number : -9223372036854775808
OUTPUT_LIST : 3
factorial : 1
EXECUTE_TESTCASE
```

analyzing characteristics of the SUVV model after the interpretation of test cases.

All defects found in these three stages are saved in a text file and compiled in a HTML table. Figures 4.10 and 4.9 show, respectively, the list of defects in a text file and its HTML table for *factorial* problem.

As it can be seen, in Figure 4.9, defects are arranged providing an overview of the number of defects found in each class. Otherwise, Figure 4.10 shows a text file that relates a more detailed list of defects found. This text file must be understood as follows:

- *Defect #*: defect counter number.
- *Testcase*: when it is set as “ALL”, it means that this defect occurred in all

Figure 4.9 - Classes of defects found in *factorial problem*.

Defect Type	Defects
Notice	-
Bad Practice	2
Performance	1
Useless Code	-
Vulnerability	-
Unreliable Code	2
<b>TOTAL</b>	<b>5</b>

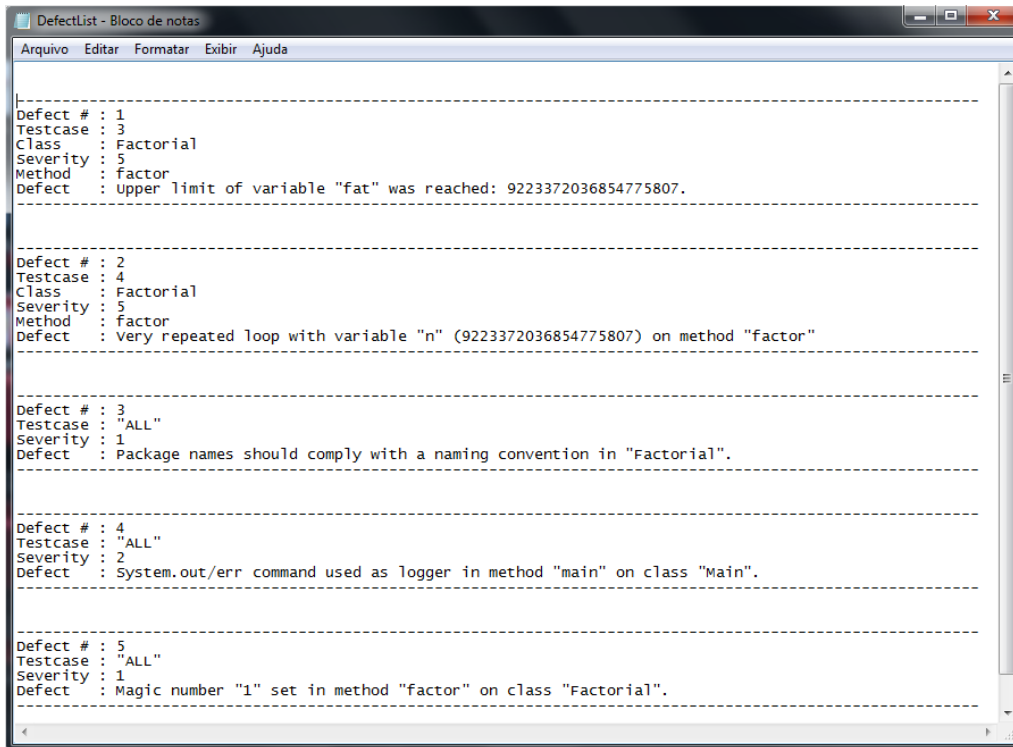
test cases. Otherwise, if it presents a list of numbers separated by comma, it means that this defect occurred just in test cases identified by these numbers.

- *Class*: it is showed if this defect occurred within a particular class. Otherwise, this field is suppressed.
- *Severity*: it shows the class of the defect by following this order: 0 - Notice, 1 - Bad Practice, 2 - Performance Loss, 3 - Useless Code, 4 - Vulnerability and 5 - Unreliable Code.
- *Method*: it is showed if this defect occurred within a particular method. Otherwise, this field is suppressed.
- *Defect*: this is a more detailed description of the defect found, very useful to locate it in the source code.

REACTOR implements the search for a particular set of defects, which were divided in six classes based on the classification used in other static analysis tools that is presented in Chapter 5. The search for other defects can be implemented and incorporated in the tool once its defective pattern is set. Table 4.9 shows the complete list of defects that can be found by REACTOR, and it can be interpreted as follows:

- *Defect*: it is a short description of each defect.
- *Class*: it is the class to which the defect type belongs.

Figure 4.10 - List of defects found in *factorial* problem.



```
DefectList - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda

-----
Defect # : 1
Testcase : 3
Class    : Factorial
Severity : 5
Method   : factor
Defect   : upper limit of variable "fat" was reached: 9223372036854775807.
-----

Defect # : 2
Testcase : 4
Class    : Factorial
Severity : 5
Method   : factor
Defect   : Very repeated loop with variable "n" (9223372036854775807) on method "factor"
-----

Defect # : 3
Testcase : "ALL"
Severity : 1
Defect   : Package names should comply with a naming convention in "Factorial".
-----

Defect # : 4
Testcase : "ALL"
Severity : 2
Defect   : system.out/err command used as logger in method "main" on class "Main".
-----

Defect # : 5
Testcase : "ALL"
Severity : 1
Defect   : Magic number "1" set in method "factor" on class "Factorial".
-----
```

- *State*: it shows in what stage this defect type can be detected.
- *Description*: it is a more detailed description of each defect type.

Table 4.9 - Defects detected by REACTOR.

#	Defect	Class	Stage	Description
1	File dependence	Notice	Model	Notifies the dependence of external files for the properly SUVV execution
2	Inner class	Notice	Model	Notifies the existence of an inner class. Should it be a inner class?
3	Close to upper limit (byte)	Unreliable Code	Model	A byte variable was set with a value dangerously close to the upper limit of this type.
4	Close to lower limit (byte)	Unreliable Code	Model	A byte variable was set with a value dangerously close to the lower limit of this type.
5	Close to upper limit (short)	Unreliable Code	Model	A short variable was set with a value dangerously close to the upper limit of this type.
6	Close to lower limit (short)	Unreliable Code	Model	A short variable was set with a value dangerously close to the lower limit of this type.
7	Close to upper limit (int)	Unreliable Code	Model	A int variable was set with a value dangerously close to the upper limit of this type.

(Continue)



Table 4.9 - Continuation

#	Defect	Class	Stage	Description
8	Close to lower limit (int)	Unreliable Code	Model	A int variable was set with a value dangerously close to the lower limit of this type.
9	Close to upper limit (long)	Unreliable Code	Model	A long variable was set with a value dangerously close to the upper limit of this type.
10	Close to lower limit (long)	Unreliable Code	Model	A long variable was set with a value dangerously close to the lower limit of this type.
11	Close to upper limit (float)	Unreliable Code	Model	A float variable was set with a value dangerously close to the upper limit of this type.
12	Close to lower limit (float)	Unreliable Code	Model	A float variable was set with a value dangerously close to the lower limit of this type.
13	Close to upper limit (double)	Unreliable Code	Model	A double variable was set with a value dangerously close to the upper limit of this type.

(Continue)

Table 4.9 - Continuation

#	Defect	Class	Stage	Description
14	Close to lower limit (double)	Unreliable Code	Model	A double variable was set with a value dangerously close to the lower limit of this type.
15	Magic number	Bad Practice	Model	The use of constant values in methods is not recommended
16	Naming convention of packages	Bad Practice	Model	Package names should comply with a naming convention
17	Naming convention of class	Bad Practice	Model	Class names should comply with a naming convention
18	Naming convention of variables	Bad Practice	Model	Variable names should comply with a naming convention
19	Naming convention of methods	Bad Practice	Model	Methods names should comply with a naming convention
20	Print command as logger	Performance Loss	Model	System.out/err command used as logger
21	Collapsed "ifs"	Performance Loss	Model	Collapsed "ifs" can be merged in source code
22	Repeated positive loop	Unreliable Code	Model	Very repeated loops until huge large positive values
23	Repeated negative loop	Unreliable Code	Model	Very repeated loops until huge large negative values
24	Useless class	Useless Code	Model	A class was implemented but is unused
25	Useless method	Useless Code	Model	A method was implemented but it is unused

(Continue)

Table 4.9 - Continuation

#	Defect	Class	Stage	Description
26	Useless constructor	Useless Code	Model	A constructor was implemented but it is unused
27	Unreachable code block	Useless Code	Model	A source code block was implemented but it is unreachable
28	Uninitialized variable	Useless Code	Path	A variable was not initialized (probably it was initialized into a not performed condition)
29	Misused library	Useless Code	Model	A library was imported but it is unused
30	SQL Injection	Vulnerability	Code Line	Unsecure SQL command execution
31	Public attribute	Vulnerability	Model	Public visibility of an unsecure attribute (it should be private or protected)
32	Public constructor	Vulnerability	Model	Public visibility of an unsecure constructor (it should be private or protected)
33	Operation handling issue	Unreliable Code	Code Line	Operations that can become an error or failure depending on input data of test cases
34	Null variable	Unreliable Code	Path	Variable that was used after set as "null"
35	Division by zero	Unreliable Code	Code Line	Operations that becomes a division by zero
36	Upper limit (byte)	Unreliable Code	Code Line	Upper limit of variable byte type reached
37	Lower limit (byte)	Unreliable Code	Code Line	Lower limit of variable byte type reached
38	Upper limit (short)	Unreliable Code	Code Line	Upper limit of variable short type reached
39	Lower limit (short)	Unreliable Code	Code Line	Lower limit of variable short type reached

(Continue)

Table 4.9 - Continuation

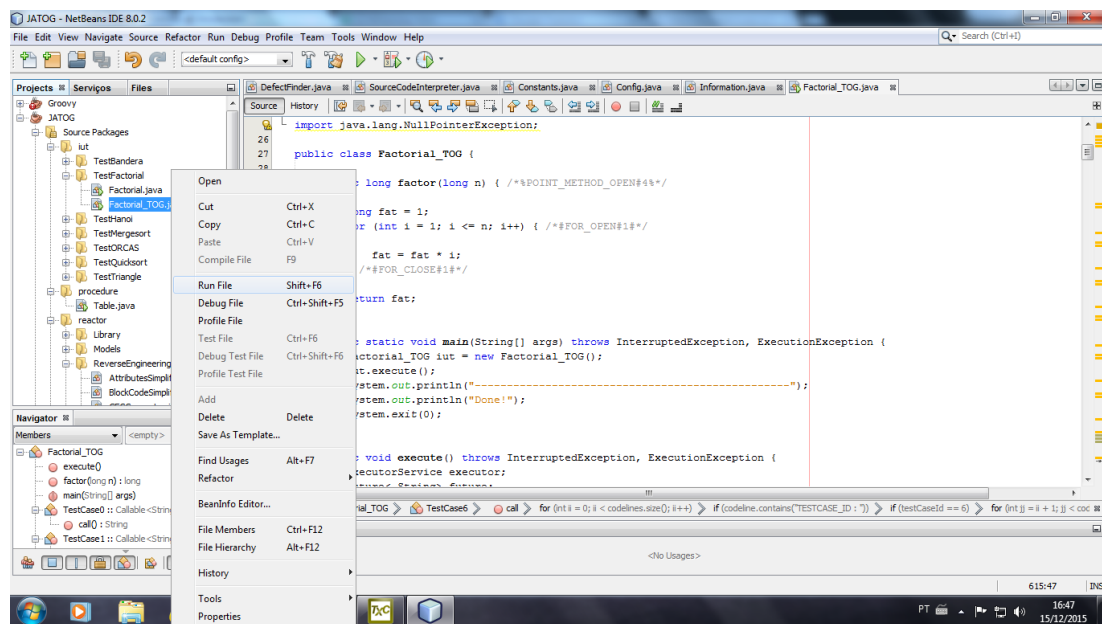
#	Defect	Class	Stage	Description
40	Upper limit (int)	Unreliable Code	Code Line	Upper limit of variable integer type reached
41	Lower limit (int)	Unreliable Code	Code Line	Lower limit of variable integer type reached
42	Upper limit (long)	Unreliable Code	Code Line	Upper limit of variable long type reached
43	Lower limit (long)	Unreliable Code	Code Line	Lower limit of variable long type reached
44	Upper limit (float)	Unreliable Code	Code Line	Upper limit of variable float type reached
45	Lower limit (float)	Unreliable Code	Code Line	Lower limit of variable float type reached
46	Upper limit (double)	Unreliable Code	Code Line	Upper limit of variable double type reached
47	Lower limit (double)	Unreliable Code	Code Line	Lower limit of variable double type reached

## 4.4 Instrumented SUVV Execution

Once expected results have been estimated, it is necessary to obtain actual results for the same set of test cases, as discussed in Section 3.5. With this intention, REACTOR instruments SUVV by creating a copy of its “main” class with the same file name followed by “\_TOG.java”. This copy is automatically changed to execute SUVV realizing the *driver* work with the same values of test cases by reading input values from the text file previously showed in Figure 4.8.

Although, this instrumented class is created automatically, it can not be executed automatically. So, at this point the testing professional must access it in a IDE (Integrated Development Environments) and execute it manually by selecting “run” option, as showed in Figure 4.11. After this execution, actual results of SUVV are also saved in text file that is read by oracle procedure.

Figure 4.11 - Running the SUVV with test cases manually.



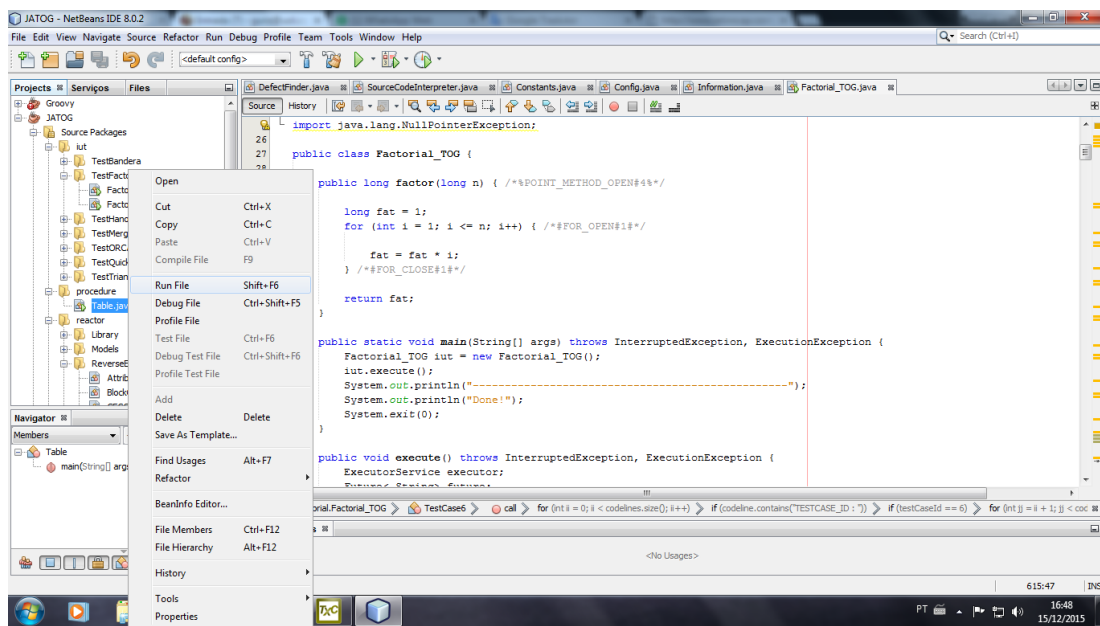
## 4.5 Oracle Procedure Execution

As explained in Section 3.5, the final verdict for a determined test case can only be inferred if an estimated expected result can be compared with an actual result. So, the procedure implemented in REACTOR gets all estimated expected results

(shown in Figure 4.8) and compares with actual results. The execution of oracle procedure also must be done manually, as presented in Figure 4.12.

The result of comparison between the estimated expected results and actual results obtained for each of test cases provides a verdict that is reported in a HTML table automatically built by REACTOR. Figure 4.13) shows the HTML table with verdicts of the *factorial problem* case study.

Figure 4.12 - Running the oracle procedure manually.



In Figure 4.13, it can be seen that from seven test cases, two of them did not obtain “pass” verdicts since they presented failures.

In test cases 0, 1, 3 and 6, the verdict is “pass” because estimated expected results and actual results were equal. It should be mentioned that the expected and actual result “1” does not represent the reality of a factorial operation with these entries. However, SUVV has this value as a result because the output variable has been set with “1” so that the tool does not return an exception when assembling the HTML table with *null* values.

The 5th test case got a “pass” verdict with both correctly computed results. The expected outputs estimated in test cases 2 and 4 were filled with “-” because the

Figure 4.13 - Test oracle generated for in *factorial problem*.

Test Case #	Inputs Generated	Outputs Expected	Outputs Obtained	Verdict
	number (long)	factorial (long)	factorial (long)	
0	-94	1	1	pass
1	0	1	1	pass
2	49	-	8789267254022766592	inconclusive
3	-9223372036854775808	1	1	pass
4	9223372036854775807	-	Exception	inconclusive
5	7	5040	5040	pass
6	-7	1	1	pass

return

tool had its run interrupted before computing the result. It occurs since REACTOR has an automatic timeout routine that interrupts test cases that take a long time to be processed. Also, note that in this same test case, the execution of the tool ed one exception and inconclusive verdicts, due to the fact that it is not possible to compare two valid results.

#### 4.6 Final Remarks

The proper configuration of REACTOR is essential to locate source code files and generate test cases based on inputs and outputs of SUVV. Such data must be set by the software tester before executing REACTOR itself. Only then, it can generate test oracle information and detect software defects only by source code analysis. The detection of defects is done by mixing static and dynamic information, revealing classes of defects that are generally unreachable via static analysis only. The oracle procedure compares the results inferred by test oracle and results obtained by a test case executor, and so REACTOR can determine the estimated verdict for each test case generated and save this report in a text file and HTML tables.





## 5 EVALUATION

In order to evaluate the approach proposed in this PhD thesis, nine classic programming problems and two real implementations were considered: *Factorial Problem*, *Triangle Classification*, *Hanoi Towers*, *Quick Sort*, *Merge Sort*, *Bubble Sort*, *Insertion Sort*, *Fibonacci Series*, *Arithmetic Mean*, *Threads* and *ORCAS* (a simulated space application software).

The number of input variables for each test case were set in a black box perspective. REACTOR’s static analysis defect detection was compared with three other known tools - *FindBugs* (FINDBUGS, 2015), *SonarQube* (SONARQUBE, 2015) and *SciTools Understand* (SCITOOLS, 2015) - which perform source code static analysis, and the success criteria considered for this work is basically the number of defects detected for each defect class.

It is important to comment that *FindBugs*, *SonarQube* and *SciTools Understand* are tools which perform static analysis. Therefore, the comparison stresses the ability to detect defects by static analysis. With respect to the detection of defects based on testing, i.e. testing defects detected by REACTOR’s oracle, we were not able to compare REACTOR with any other approach/tool. Although there are several automated testing oracle efforts in the literature, it was not possible to identify a solution that addresses this problem considering only the source code as proposed by REACTOR.

### 5.1 Evaluation Criteria

There is no standard for classifying software static analysis defects, and the three static analysis tools have differences in this respect. *Understand* does not have any classification type of defects. *SonarQube* classifies the defects clearly based on its severity (“Blocker”, “Critical”, “Major”, “Minor” and “Info”). And finally, *FindBugs* classifies based on the technical aspect of defect (“Bad practice”, “Correctness”, “Experimental”, “Internationalization”, “Malicious code vulnerability”, “Performance”, “Security” and “Dodgy code”).

In REACTOR, it was decided to make a technical classification related to the defect type, but also somehow convey an immediate idea of its severity. Some classes were based on *FindBugs* and other new classes were created. This classification is detailed in Section 3.4. Therefore, as there is no established standard to classify all defects detected for all tools, the authors conducted an *ad hoc* analysis of each defect in

order to arrange them within REACTOR classes.

## 5.2 Overall Results

Figures 5.1 and 5.2 show the results of the static defects detection comparing REACTOR's static code analysis feature with other three tools. In these tables, we report only the **true positive static defects**. Figure 5.3 shows the results of REACTOR's testing capability.

In Figures 5.1 and 5.2, gray column exhibits the case study. Orange column contains the six classes of defects that are being compared. Yellow column shows the number of defects detected by REACTOR within each class, and white columns show the defects detected by other tools. And finally, green row presents the sum of defects detected by each tested tool in each case study.

In Figure 5.3, gray column exhibits the number of test cases generated (number within parenthesis is the original number before being reduced by pairwise approach). The number of test case verdicts is in yellow columns separated by “*pass*”, “*no pass*” and “*inconclusive*”, and the last column presents the number of defects that influenced the negative and inconclusive verdicts.

### 5.2.1 Case Study 1: Factorial Problem

In *factorial problem*, whose source code is used as running example in Chapter 3, REACTOR was tested using as input and output two numeric values which are the numbers that must be calculated by  $n!$  and its result respectively. Seven test cases were automatically generated and a failure was found when SUVV was submitted with the upper limit of its numerical input variable.

Note that only REACTOR found unreliable code lines, whose result presented two “*inconclusive*” verdicts in this case study, which was presented in Chapter 3.

### 5.2.2 Case Study 2: Hanoi Towers

This is a very common programming problem whose source code is presented in Section B.1. It consists of three fixed towers with an input variable that corresponds to the number of discs that must be moved, and an output variable that is the number of movements performed. Seven test cases were automatically generated and two failures were discovered resulting in a “*no pass*” and an “*inconclusive*” verdict.

Figure 5.1 - REACTOR's true positive static defects detection compared with other tools (Part I).

Case Study	Defect Class	REACTOR	Find Bugs	Under stand	Sonar Qube
Factorial Problem	Notice	-	-	-	-
	Bad Practice	2	-	-	2
	Performance Loss	1	-	-	1
	Useless Code	-	-	-	-
	Vulnerability	-	-	-	1
	Unreliable Code	2	-	-	-
	<b>TOTAL</b>	<b>5</b>	<b>0</b>	<b>0</b>	<b>4</b>
Hanoi Towers	Notice	-	-	-	-
	Bad Practice	8	-	-	5
	Performance Loss	1	-	-	1
	Useless Code	2	-	-	-
	Vulnerability	1	-	-	1
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>12</b>	<b>0</b>	<b>0</b>	<b>7</b>
Triangle Classification	Notice	-	-	-	-
	Bad Practice	6	4	-	9
	Performance Loss	-	-	-	-
	Useless Code	2	1	1	1
	Vulnerability	3	-	-	4
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>11</b>	<b>5</b>	<b>1</b>	<b>14</b>
Quick Sort	Notice	-	-	-	-
	Bad Practice	5	-	-	2
	Performance Loss	9	1	2	9
	Useless Code	3	-	-	-
	Vulnerability	-	1	-	1
	Unreliable Code	2	-	-	-
	<b>TOTAL</b>	<b>19</b>	<b>2</b>	<b>2</b>	<b>12</b>
Merge Sort	Notice	-	-	-	-
	Bad Practice	6	-	-	-
	Performance Loss	8	1	1	-
	Useless Code	6	-	-	-
	Vulnerability	-	1	-	-
	Unreliable Code	2	-	-	-
	<b>TOTAL</b>	<b>22</b>	<b>2</b>	<b>1</b>	<b>0</b>
Bubble Sort	Notice	-	-	-	-
	Bad Practice	2	-	-	2
	Performance Loss	1	1	1	1
	Useless Code	3	-	-	-
	Vulnerability	-	-	-	-
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>6</b>	<b>1</b>	<b>1</b>	<b>4</b>

Figure 5.2 - REACTOR's true positive static defects detection compared with other tools (Part II).

Case Study	Defect Class	REACTOR	Find Bugs	Under stand	Sonar Qube
Insertion Sort	Notice	-	-	-	-
	Bad Practice	2	-	-	2
	Performance Loss	1	1	1	1
	Useless Code	3	-	-	-
	Vulnerability	-	-	-	1
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>6</b>	<b>0</b>	<b>0</b>	<b>5</b>
Fibonacci Series	Notice	-	-	-	-
	Bad Practice	1	1	-	1
	Performance Loss	-	-	-	-
	Useless Code	-	-	-	-
	Vulnerability	-	-	-	-
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>
Arithmetic Mean	Notice	-	-	-	-
	Bad Practice	3	-	-	3
	Performance Loss	1	1	-	1
	Useless Code	2	-	-	-
	Vulnerability	-	-	-	1
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>6</b>	<b>1</b>	<b>0</b>	<b>5</b>
Threads	Notice	-	-	-	-
	Bad Practice	8	-	-	9
	Performance Loss	-	-	-	-
	Useless Code	15	4	1	1
	Vulnerability	3	-	-	4
	Unreliable Code	-	-	-	-
	<b>TOTAL</b>	<b>26</b>	<b>4</b>	<b>1</b>	<b>14</b>
ORCAS	Notice	4	-	4	1
	Bad Practice	102	3	-	85
	Performance Loss	7	11	6	4
	Useless Code	51	3	12	8
	Vulnerability	24	1	-	16
	Unreliable Code	1	-	-	-
	<b>TOTAL</b>	<b>189</b>	<b>18</b>	<b>22</b>	<b>114</b>

Figure 5.3 - REACTOR's testing defects detection.

Case Study	Test Cases	Verdicts			Failures
		Pass	No Pass	Inc.	
Factorial Problem	7	5	0	2	2
Hanoi Towers	7	5	1	1	2
Triangle Classification	84 (343)	84	0	0	0
Quick Sort	7	2	4	1	5
Merge Sort	7	3	0	4	4
Bubble Sort	7	3	0	4	4
Insertion Sort	7	3	0	4	4
Fibonacci Sequence	7	6	1	0	1
Arithmetic Mean	49	46	3	0	3
Threads	49	28	0	21	21
ORCAS	49	48	0	1	1

In case of detecting useless code as a defect, REACTOR performed better than other tools. REACTOR is specially useful in order to detect such defect, since an unreachable code block can be perceived by REACTOR because it performs real test cases in order to exercise a path between code blocks, unlike other tools that detect unreachable code only if it is a method, constructor or class that is not used.

### 5.2.3 Case Study 3: Triangle Classification

*Triangle Classification Problem* has its source code presented in Section B.2, and it consists of three input numeric variables corresponding to the three sides of the triangle, and an output text variable describing triangle's type that can be "equilateral", "isosceles" or "scalene". For this case study, 343 test cases were automatically generated by combining input variables, and then they were reduced to 84 by pairwise algorithm.

In this case study it can be seen that all test cases passed. And this was the only case study that the tool (*SonarQube*) found more defects than REACTOR. However, most of these defects are related just to bad coding practices of minor importance. And again, REACTOR was able to detect more useless code parts than other tools. No failure was reported in this case study.

### 5.2.4 Case Study 4: Quick Sort

Sorting algorithms are very useful in computer science, and *Quick Sort* is a very commonly used algorithm for sorting, since it is usually faster than any other algorithm for this purpose (SKIENA, 2008). This problem consists basically in sorting an array of random numeric values. The input, in this case, is the array size that is randomly generated, and the output is the number of elements that were not sorted in array at the end of execution. The source code of this case study is presented in Section B.3.

REACTOR was much better than the other 3 tools when compared with respect to detection of defects. Beyond two unreliable code lines, it detected 10 useless code parts (variables, blocks, methods, constructors or classes) that were not exercised by test cases, and no other tool was able to find even one. Five failures were discovered and reported as four "no pass" and one "inconclusive" verdicts.

### 5.2.5 Case Study 5: Merge Sort

*Merge Sort* is a sorting algorithm that implements the approach of “divide and conquer”. It uses recursion to reduce large problems into smaller ones by partitioning the set of elements into two smaller groups, and sorting each one of these recursively (SKIENA, 2008). As with *Quick Sort* case study, the input is an integer that determines the array size which may be randomly generated, and the output is the number of unsorted elements at the end of execution. The source code of this case study is presented in Section B.4.

Seven test cases were executed and several defects were found by REACTOR that were not found by other tools, mainly, useless code, performance loss and unreliable code parts that may return into failures. Four failures, that were reported as “*inconclusive*” verdicts, were discovered in this case study.

### 5.2.6 Case Study 6: Bubble Sort

Also known as *Sinking Sort*, *Bubble Sort* is a sorting algorithm that is based on the swaps of values that are in the wrong order. This algorithm is known as “bubble” due to the way that smaller values go to the top of the list like bubbles. As with *Quick Sort* and *Merge Sort*, in this case study the input is an integer value which sets the array size that may be randomly generated, and the output is the number of unsorted elements at the end of execution. The source code of this case study is presented in Section B.5.

Three useless code defects were detected by REACTOR and undetected by other tools, once seven test cases were generated and executed. However, *SonarQube* detected one more performance loss defects due to an unnecessary collapsed “if” which REACTOR does not detected, since its within a code block that was not performed (and so, it is reported as useless code). Three “*pass*” and four “*inconclusive*” verdicts were reported due to defects found by REACTOR.

### 5.2.7 Case Study 7: Insertion Sort

*Insertion Sort* is a simple sorting algorithm that basically builds an array by inserting sorted values one at a time. This algorithm is simple to implement, however, it is much less efficient on large arrays than other sorting algorithms. As with the others already presented as case studies, the input of *Insertion Sort* is an integer value which corresponds to the array size that is randomly generated, and the number of unsorted elements at the end of execution is the output. The source code of this

case study is presented in Section B.6.

This case study presented similar defects if compared to *Bubble Sort* algorithm, and it occurred due to the fact that both the implemented source codes present a very similar structure.

### 5.2.8 Case Study 8: Fibonacci Series

The *Fibonacci Series* is one of the most known problems discussed in computing and mathematics, and it is a set of numbers that starts with a one (or a zero followed by a one), and proceeds based on the rule that each number (named a *Fibonacci Number*) is equal to the sum of the preceding two numbers. For this case study, the input is the series size, and the output is the next value of the *Fibonacci Series*. The source code of this case study is presented in Section B.7.

This case study is the one with lower occurrence of defects detected among test cases, both by REACTOR and the others. However, even with few defects detected, one of the seven test cases was reported as “no pass” since the execution of one test case reached the time limit stipulated by REACTOR and so the output variables was not set.

### 5.2.9 Case Study 9: Arithmetic Mean

In mathematics, the arithmetic mean is the sum of a certain quantity of numbers divided by that quantity. The source code of this case study is presented in Section B.8 and it is used by (MARINKE, 2012), which basically calculates the mean grade of a certain number of semesters, supposing that each semester is represents by one grade.

By combining input variables, 49 test cases were generated of which 3 were reported as “no pass”. Once again, REACTOR detected 2 useless code parts that others did not. However, *SonarQube* detected a vulnerability related to the existence of a public constructor that is not explicitly implemented, but it seems that it is automatically generated by Java compiler.

### 5.2.10 Case Study 10: Threads

The source code of this case study is presented in Section B.9, it was used in Marinke (2012) and simulates a pipelined computation, where each pipeline stage is executed as a separate thread. The stages interact through a “Connector” object that imple-



ments typical methods for handling data. Forty nine test cases were generated from the combination of two sets of seven testing values for integer.

The number of defects found shows that REACTOR was far better than other tools particularly with respect to useless code parts, as it has become typical in REACTOR. No failures were reported in this case study as “*no pass*” verdicts, although a huge number of failures reported as “*inconclusive*” verdicts were discovered.

### 5.2.11 Case Study 11: ORCAS

The ORCAS (*Observação de Raios Cósmicos Anômalos e Solares na Magnetosfera*) (INSTITUTO..., 1998) software is a real scientific experiment developed to be embedded on satellites, and it can be considered as an industry case study. The test case generator combined two sets of seven values for byte type testing, and it generated forty nine test cases for analysis and execution.

In this case study, one more time REACTOR detected more defects than other tools including several bad practices, performance losses, useless code, vulnerabilities and an unreliable code part. The results obtained by this case study are specially interesting, since it demonstrates that the approach implemented in REACTOR can handle real world applications.

## 5.3 Additional Discussion About the Evaluation

With respect to static code analysis, in the 1st, 3rd, 4th, 5th, 7th, 9th, 10th and 11th case studies, other tools detected vulnerabilities that REACTOR did not. In the 1st, 3rd, 4th, 7th, 9th and 10th, REACTOR did not find specifically a method with public visibility, which is characterized as a vulnerability issue in *SonarQube*. In fact there is no public method explicitly implemented, but as the empty constructor was not implemented, the tool *SonarQube* probably foresees that the Java compiler automatically creates an empty public constructor and treats this issue as a vulnerability. In the 3th case study, REACTOR did not find three attributes with public visibility, which is also classified as an vulnerability issue. These types of defects are implemented in REACTOR, and their implementation will be inspected in order to fix it hereafter. Also, in the 4th and 5th case study, *Findbugs* tagged as vulnerability the passing of a changeable size object through method’s parameter.

Other defects that REACTOR could not find are in 3rd and 10th case studies, where *SonarQube* found more bad practice defects. These defects found by *SonarQube* are related to parameter names (in a method) that do not comply with a naming con-

vention. Finally, in the 11th case study, *FindBugs* found more performance losses than REACTOR, and that happened because *FindBugs* detected some calls of deprecated methods implemented in JDK. That type of defect is still not implemented in REACTOR yet, however, as with other defects that it did not find, it is planned to be implemented hereafter.

In most of the case studies, REACTOR detected more defects than the other three tools except for one of them. This happened even if REACTOR is not able to detect the same variety of defects of these established tools (the current implementation of REACTOR detects 47 types of static analysis defects, and *FindBugs*, for example, is able to detect about 400 types of static analysis defects). However, the combination of static and dynamic techniques into a single approach resulted in a more favorable REACTOR's performance.

Figure 5.4 shows that, in general, REACTOR detected more true positive static defects than the other three tools when compared to the total amount of defects which were detected in each case study. All defects detected by all tools were considered as the total number of defects, with the exception of the same defects that were detected by more than one tool, that represents in practice one defect only. In other words, the rate presented in Figure 5.4 is a *recall* where:

$$recall = \frac{\# \text{ relevant defects}}{\text{total relevant defects}}$$

As it can be seen, REACTOR detected more than 90% of the total relevant defects presented in case studies.

One known problem related to static code analysis tools is the huge amount of false positives. The fact that a professional does not agree with certain rule implemented in a static analysis tool not necessarily mean that there is a false positive. However, results by using static analyzers are usually very noisy where there may not only have false positives but also many defects identified that are not, in fact, important to a particular software product or coding standard. These points are some drawbacks regarding the use of static analysis tools.

Although it is not possible to assert that false positives were detected by running

Figure 5.4 - Rate of true positive static defects.

Defect Class	Total of Defects	Rate of Defect Detection			
		REACTOR	FindBugs	Understand	SonarQube
1 - Factorial Problem	6	83.33%	0%	0%	66.67%
2 - Hanoi Towers	12	100%	0%	0%	58.33%
3 - Triangle Classification	15	73.33%	33.33%	6.67%	93.33%
4 - Quick Sort	20	95%	10%	10%	60%
5 - MergeSort	23	95.65%	8.7%	4.35%	0%
6 - Bubble Sort	6	100%	16.67%	16.67%	50%
7 - Insertion Sort	8	75%	0%	0%	62.5%
8 - Fibonacci Series	1	100%	100%	0%	100%
9 - Arithmetic Mean	7	85.71%	14.29%	0%	71.43%
10 - Threads	28	92.86%	14.29%	3.57%	50%
11 - ORCAS	193	97.98%	9.33%	11.4%	59.07%
<b>RECALL (%)</b>		<b>90.81%</b>	<b>18.78%</b>	<b>4.79%</b>	<b>61.03%</b>

*FindBugs*, *Understand*, and *SonarQube*, some unimportant results, in *Understand* and *SonarQube*, were identified.

Some issues that these tools identify as defects were considered unimportant because they do not present any relevance for testing, so they were suppressed from results. For example, *Understand* lists as defects methods that do not have an exit point or methods that have more than one exit point (in practice the exit point is the “return” in Java). Other examples are *SonarQube* which considers as a defect annotations located at the end of code lines, and *FindBugs* has a spell checker for annotations which considers misspellings as defects, among others. Such occurrences may be obstacles in the real word settings, since one of the major criticisms of static analysis is that these tools often return reports with a lot of information (i.e., unimportant defects), discouraging their use by testers. So, REACTOR has planned to does not detect defects considered unimportant.

Figure 5.5 presents the percentage of unimportant static analysis defects detected by running these two tools for the first six case studies (Figure 5.1). This percentage is calculated as follows:

$$unimp = \frac{\# \text{ unimportant}}{(\# \text{ true positive}) + (\# \text{ unimportant})}$$

Figure 5.5 - Rate of unimportant defects.

Test Case	Rate of Unimportant Defects	
	Understand	SonarQube
1 - Factorial Problem	100%	33%
2 - Hanoi Towers	100%	36%
3 - Triangle Classification	75%	0%
4 - Quick Sort	94%	20%
5 - MergeSort	98%	0%
6 - Bubble Sort	95%	0%
<b>Average (%)</b>	<b>93.63%</b>	<b>14.93%</b>

According to the type of static defects defined in this PhD thesis, it was not detected unimportant defects neither in REACTOR nor in *FindBugs*. However, as shown in Figure 5.4, REACTOR detected many more true positive static defects than *FindBugs*.

Regarding the testing capability, except for the 3rd case study, REACTOR found defects that resulted in failures in all of them. As previously mentioned, it was not possible to compare REACTOR with other test oracle approaches because it was not found, in the literature, other test oracle solutions which work based only on the source code as REACTOR.

#### 5.4 Final Remarks

The capability of REACTOR to detect software defects was compared with three well known static analysis tools in order to validate the approach proposed in this work. For this purpose, 11 case studies were assessed, being 9 of them related to the classical computing problems, and 2 of them real applications. The 11th case study can be considered an industry case study. REACTOR presented better results in most of the case studies. Next chapter presents conclusions and future directions of this work.

## 6 CONCLUSION

The main advantage of an automated static analysis approach is to enable a significant reduction in the cost of revealing software defects, which drastically reduces the cost of development for the entire project. Unlike manual inspections, the automated static analysis can have a full code coverage by checking routines even when very rarely used, and that are hardly checked by programmers. So, it can reveal defects that may not manifest as failures for a long time. However, unlike other testing techniques, static analysis is unable to detect any kind of defects directly related to the dynamic information handled by source code, as numeric overflows or exceptions. Such defects can only be addressed by executing the SUVV, or at least part of it.

Therefore, like any other testing methodology, static analysis has a lot of strong points but it also has its weaknesses, and its weak points certainly reinforces the *cliche* that there is no “silver bullet” in software testing. Different methodologies always produce different results for different approaches. Thus, once there is no perfect testing method, the proper combination of more than one methodology joining their strong points can be an alternative to achieve a higher software quality.

The development of high quality software products is nowadays essential. The issues in software may have minor but also major consequences such as causing great financial costs or risks to human lives. Thus, combining different V&V strategies into a single approach is interesting because it is possible to benefit from the potentials of each method in an integrated solution.

The REACTOR method and its implementation are in line with this reasoning where static code analysis, testing, and reverse engineering are employed aiming to address a wider range of software defects. With respect to static code analysis, REACTOR uses dynamic information in order to complement this analysis and the results presented in this PhD thesis confirm it is a promising solution.

The methodologies developed to support automated software testing activities generally uses high level specifications as a means to computationally handle abstraction. However, high level specifications, for most of the cases, are not detailed enough to be able to detect defects and prevent failures by purely analyzing them. Also, developing and maintaining several specifications with a high degree of details can raise costs prohibitively. And in order to outline this, the approach proposed in REACTOR does not require such specifications and works based only on the source

code.

REACTOR was compared with three well known open source and commercial static code analyzers. To sum up, considering all evaluated case studies, including one complex case study for the space domain, REACTOR performed better, based on the amount of detected true positive static defects, than the other tools, with the exception of the *Triangle Classification Problem*, where *SonarQube* was slightly better, and *Fibonacci Series* where it can be considered a draw. In all other case studies, REACTOR was better showing the benefits of our proposal.

Concerning the amount of unimportant defects detected, none was found by using REACTOR. This is not the case considering *Understand* and *SonarQube* where the average value of unimportant static analysis defects was 93.63% and 14.93%, respectively, for the first six case studies. Although it did not also find unimportant results via *FindBugs*, REACTOR detected many more true positive static defects than *FindBugs*. This is due to the combination of static and dynamic analysis. REACTOR is suitable to detect useless code as other tools do not. This is because the combination used in the thesis finds, not only methods or classes not used, but also code blocks that are not exercised by any test case. Only the static analysis by itself is not enough to reveal this kind of defect in that level of detail.

For testing, REACTOR generates test cases according to four different techniques: BVA, EP, RT, and combinatorial designs (pairwise testing). By mixing these techniques, a new technique called AEBPA was created. To the best of our knowledge, no other automated test oracle approach in the literature addresses this problem based only on the source code as this work proposes.

## 6.1 Requirements and Limitations

Due to its characteristics, this approach depends on some requirements and presents some limitations.

As mentioned in Section 4.3.1, the execution of REACTOR requires a description of input and output variables along with their types. Therefore, the testing professional must know this information based on the SUVV. In addition, these variables must be only primitive types since the oracle procedure can not compare instances of classes.

SUVV must have only one *main* method, so that REACTOR knows where source code interpretation must start, and, it must be written in Java. REACTOR works

fully based only on source code analysis, therefore, modifications to use other languages as C++, PHP or C# are totally possible, but not quite trivial.

Before the SUVV is executed with the set of test cases for obtaining the actual results, it must have been compiled without any errors.

To minimize the probability of an error in identifying code lines, it is recommended that the testing professional makes a proper formatting of the source code before running REACTOR. The most of IDE's for Java, such as NetBeans ([NETBEANS, 2015](#)) or Eclipse ([ECLIPSE, 2015](#)), have this functionality available.

Finally, SUVV should not implement any GUI, since GUI testing requires other different approaches to somehow simulate user interactions.

## **6.2 Future Work**

Future directions include the improvement of REACTOR in order to bypass some of the limitations discussed in Section 6.1. A very important task is the development of new types of defects that may be addressed by static analysis.

Other interesting possibilities are the improvement of REACTOR's usability by the implementation of a friendly GUI, and the development of features to perform static analysis of SUVV coded in other programming languages such as C++, C# and PHP.

The application of REACTOR in other domains is another effort to pursue, as the increment of its capability to infer expected results by test oracle using other combines techniques as AI (Artificial Intelligence) based, for example.

## **6.3 Final Remarks about this PhD Thesis**

This thesis presents a new approach that combines static and dynamic analyzes to automate the solution of two complex problems: detection of defects and oracle problem. In order to prove that this approach is feasible, it was implemented on an experimental tool that presented good results compared to other known tools that perform almost similar task.

Although an experimental tool and without GUI, REACTOR is not complicated to be used. It requires basically, from tester, identification of the names of the input variables, output variables, and directories where the SUVV is located. This factor is very encouraging for the use of this tool in practical context. In addition, REACTOR

fills an important gap in academia as it focuses on the automatic generation of test oracles, presenting a high level of automation for testing.

According to the literature review presented in this work, no related research has the same level of automation as REACTOR in order to solve the oracle problem based only on the source code, with the additional benefit of performing a very effective detection of defects by a combination of static and dynamic analyses, as shown by the case studies.

Pragmatically, it is expected that the approach developed in this thesis can be used effectively in academic and industrial environments and henceforth contribute to improves overall SUVV quality.

May this thesis be also a new contribution to disseminate the knowledge about combined approaches that use testing oracle and static code analysis capable of detecting defects, and techniques to make them more systematic, automatic, and less resources demanding.



## REFERENCES

- AGGARWAL, A.; JALOTE, P. Integrating static and dynamic analysis for detecting vulnerabilities. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 30., 2006, Chicago. **Proceedings...** Washington: IEEE Computer Society, 2006. p. 343–350. Available from: <<http://dx.doi.org/10.1109/COMPSAC.2006.55>>. 14, 18, 28, 71
- AGGARWAL, K. K.; SINGH, Y.; KAUR, A.; SANGWAN, O. P. A neural net based approach to test oracle. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 29, n. 3, p. 1–6, may 2004. ISSN 0163-5948. Available from: <<http://doi.acm.org/10.1145/986710.986725>>. 4, 29
- ARANTES, A. O.; SANTIAGO, V. A.; VIJAYKUMAR, N. L. On proposing a test oracle generator based on static and dynamic source code analysis. In: IEEE INTERNATIONAL WORKSHOP ON MODEL-BASED VERIFICATION VALIDATION (MVV), IEEE INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY, RELIABILITY AND SECURITY, 5. (QRS), 3-5 Aug, Vancouver, Canadá. **Proceedings...** [S.l.], 2015. Setores de Atividade: Pesquisa e desenvolvimento científico. Access in: 31 maio 2016. 33
- ARCURI, A.; IQBAL, M. Z.; BRIAND, L. Black-box system testing of real-time embedded systems using random and search-based testing. In: INTERNATIONAL CONFERENCE ON TESTING SOFTWARE AND SYSTEMS (ICTSS), 22., 2010, Natal, Brazil. **Proceedings...** Berlin, Heidelberg: Springer-Verlag, 2010. (ICTSS'10), p. 95–110. ISBN 3-642-16572-9, 978-3-642-16572-6. Available from: <<http://dl.acm.org/citation.cfm?id=1928028.1928036>>. 29
- AYEWAH, N.; PUGH, W.; MORGENTHALER, J. D.; PENIX, J.; ZHOU, Y. Evaluating static analysis defect warnings on production software. In: ACM SIGPLAN-SIGSOFT WORKSHOP ON PROGRAM ANALYSIS FOR SOFTWARE TOOLS AND ENGINEERING, 7., 2007, San Diego, California, USA. **Proceedings...** New York, NY, USA: ACM, 2007. (PASTE '07), p. 1–8. ISBN 978-1-59593-595-3. Available from: <<http://doi.acm.org/10.1145/1251535.1251536>>. 29
- BALERA, J. M.; SANTIAGO JÚNIOR, V. A. T-tuple reallocation: an algorithm to create mixed-level covering arrays to support software test case generation. In: INTERNATIONAL CONFERENCE OF COMPUTATIONAL SCIENCE AND ITS APPLICATIONS (ICCSA), 15., 2015, Banff, AB, Canada. **Proceedings...**

Banff, AB, Canada: Springer International Publishing, 2015. p. 503–517. ISBN 978-3-319-21410-8. Available from:

<[http://dx.doi.org/10.1007/978-3-319-21410-8\\_39](http://dx.doi.org/10.1007/978-3-319-21410-8_39)>. 37

BALL, T. The concept of dynamic analysis. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 24, n. 6, p. 216–234, oct. 1999. ISSN 0163-5948.

Available from: <<http://doi.acm.org/10.1145/318774.318944>>. 18

BALZAROTTI, D.; COVA, M.; FELMETSGER, V.; JOVANOVIC, N.; KIRDA, E.; KRUEGEL, C.; VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY (SP 2008), 2008, Oakland, CA. **Proceedings...** Oakland, CA: IEEE, 2008. p. 387–401. ISSN 1081-6011. 28

BEYDEDA, S.; GRUH, V.; STACHORSKI, M. A graphical class representation for integrated black- and white-box testing. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2001, Florence.

**Proceedings...** Florence: IEEE, 2001. p. 706–715. ISSN 1063-6773. 2

BINDER, R. V. **Testing object-oriented systems: models, patterns and tools**. Boston, MA: Addison-Wealey, 2000. 3, 4, 9, 14, 26, 27

BOEHM, B. W. **Software engineering economics**. 1. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981. ISBN 0138221227. 10

BOYAPATI, C.; KHURSHID, S.; MARINOV, D. Korat: atomated testing based on java predicates. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2002, Roma, Italy. **Proceedings...** New York, NY, USA: ACM, 2002. (ISSTA '02), p. 123–133. ISBN 1-58113-562-9. Available from: <<http://doi.acm.org/10.1145/566172.566191>>. 30

CATSOULIS, J. **Designing embedded hardware**. [S.l.]: O'Reilly Media, Inc., 2005. ISBN 0596007558. 1

CHATZIELEFTHERIOU, G.; KATSAROS, P. Test-driving static analysis tools in search of C code vulnerabilities. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE WORKSHOPS (COMPSACW), 35., 2011, Munich. **Proceedings...** Munich: IEEE, 2011. p. 96–103. 28

CHEN, P. **The entity-relationship model: toward a unified view of data**. Cambridge, USA: Massachusetts Institute of Technology, 1976. 25

CHEN, T. Y.; TSE, T. H.; ZHOU, Z. Q. Fault-based testing without the need of oracles. **Information and Software Technology**, v. 45, p. 1–9, 2003. 30

CHESS, B.; WEST, J. **Secure programming with static analysis**. 1. ed. [S.l.]: Addison-Wesley Professional, 2007. ISBN 9780321424778. 10, 12, 13, 14

CORNELISSEN, B.; ZAIDMAN, A.; DEURSEN, A. van; MOONEN, L.; KOSCHKE, R. A systematic survey of program comprehension through dynamic analysis. **Software Engineering, IEEE Transactions on**, v. 35, n. 5, p. 684–702, Sept 2009. ISSN 0098-5589. 18

DURAN, J. W.; NTAFOSS, S. A report on random testing. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 5., 1981, Piscataway, NJ, USA. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 1981. (ICSE '81), p. 179–183. ISBN 0-89791-146-6. Available from: <<http://dl.acm.org/citation.cfm?id=800078.802530>>. 22

ECLIPSE. **Eclipse**. 2015. Available from: <<https://eclipse.org/>>. Access in: 18 mar 2015. 125

ELLSBERGER, I.; HOGREFE, D.; SARMA, A. **SDL: formal object-oriented language for communicating systems**: toward a unified view of data. [S.l.]: Prentice Hall, 1997. 26

EMANUELSSON, P.; NILSSON, U. A comparative study of industrial static analysis tools. In: INTERNATIONAL WORKSHOP ON SYSTEMS SOFTWARE VERIFICATION (SSV), 3., 2008. **Proceedings...** 2008. v. 217, p. 5 – 21. ISSN 1571-0661. Available from: <<http://www.sciencedirect.com/science/article/pii/S1571066108003824>>. 28

ERNST, M. D. Static and dynamic analysis: synergy and duality. In: **WODA 2003: ICSE Workshop on Dynamic Analysis**. Portland, OR: IEEE, 2003. p. 24–27. 13, 14

FACTORIAL. **Factorial problem**: factorial calculator (n!). 2015. Available from: <<http://www.calculatorsoup.com/calculators/discretemathematics/factorials.php>>. Access in: 18 mar 2015. 33, 36

FINDBUGS. **FindBugs™**: find bugs in java programs. University of Maryland: [s.n.], 2015. Available from: <<http://findbugs.sourceforge.net>>. Access in: 18 mar 2015. 15, 111

GANE, C.; SARSON, T. **Structured systems analysis: tools and techniques**. [S.l.]: Prentice Hall, 1979. Englewood Cliffs, N.J. 25

GAROUSI, V.; ZHI, J. A survey of software testing practices in canada. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 86, n. 5, p. 1354–1376, may 2013. ISSN 0164-1212. Available from:  
<<http://dx.doi.org/10.1016/j.jss.2012.12.051>>. 2, 20

GODEFROID, P.; KLARLUND, N.; SEN, K. Dart: directed automated random testing. **SIGPLAN Not**, ACM, New York, NY, USA, v. 40, n. 6, p. 213–223, jun. 2005. ISSN 0362-1340. Available from:  
<<http://doi.acm.org/10.1145/1064978.1065036>>. 21, 35

GRILO, A. M. P.; PAIVA, A. C. R.; FARIA, J. P. Reverse engineering of GUI models for testing. In: IBERIAN CONFERENCE ON INFORMATION SYSTEMS AND TECHNOLOGIES (CISTI), 5., 2010. [S.l.], 2010. p. 1–6. ISSN 2166-0727. 2, 12

GUTJAHR, W. J. Partition testing vs. random testing: the influence of uncertainty. **IEEE Transactions on Software Engineering**, v. 25, n. 5, p. 661–674, Sep 1999. ISSN 0098-5589. 22

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, p. 237–274, 1987. 26

HARMAN, M.; KIM, S. G.; LAKHOTIA, K.; MCMINN, P.; YOO, S. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION, AND VALIDATION WORKSHOPS (ICSTW), 3., 2010, Washington, DC. **Proceedings...** [S.l.], 2010. p. 182–191. 30

HOWDEN, W. E.; EICHHORST, H. P. Proving properties of programs from program traces. New York, USA, p. 46, 1978. Tutorial: Software Testing and Validation Techniques. : IEEE Computer Society Press. 26

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE). 610.12-1990: IEEE standard glossary of software engineering terminology. **IEEE Std 610.12-1990**, New York, NY, USA, p. 83, 1990. 9, 10

INSTITUTO NACIONAL DE PESQUISS ESPACIAIS (INPE). **EXP-OBDDH communication protocol definition: a case study for PLAVIS**, INPE Internal Publication/QSEE Project, São José dos Campos, 1998. 119

JIN, H.; WANG, Y.; CHEN, N.; GOU, Z.; WANG, S. Artificial neural network for automatic test oracles generation. In: INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING, 2008, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. (CSSE '08, v. 2), p. 727–730. ISBN 978-0-7695-3336-0. Available from: <<http://dx.doi.org/10.1109/CSSE.2008.774>>. 29

JOVANOVIC, N.; KRUEGEL, C.; KIRDA, E. Precise alias analysis for static detection of web application vulnerabilities. In: WORKSHOP ON PROGRAMMING LANGUAGES AND ANALYSIS FOR SECURITY, 30., 2006, Ottawa, Ontario, Canada. **Proceedings...** New York, NY, USA: ACM, 2006. (PLAS '06), p. 27–36. ISBN 1-59593-374-3. Available from: <<http://doi.acm.org/10.1145/1134744.1134751>>. 28

JUNIT. **Resources for test driven development**. 2015. Available from: <<http://www.junit.org/>>. Access in: 25 nov. 2015. 4

LANZA, M. **Object-oriented reverse engineering coarse-grained, fine-grained, and evolutionary software visualization**. Thesis (PhD in Computer Science and Applied Mathematics) — University of Bern, May 2003. 3

LAPIERRE, S.; MERLO, E.; SAVARD, G.; ANTONIOL, G.; FIUTEM, R.; TONELIA, P. Automatic unit test data generation using mixed-integer linear programming and execution trees. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), 1999. **Proceedings...** [S.l.], 1999. p. 189–198. ISSN 1063-6773. 29

LEI, Y.; TAI, K.-C. In-parameter-order: a test generation strategy for pairwise testing. In: INTERNATIONAL SYMPOSIUM ON HIGH-ASSURANCE SYSTEMS ENGINEERING, 3., 1998, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1998. (HASE '98), p. 254–261. ISBN 0-8186-9221-9. Available from: <<http://dl.acm.org/citation.cfm?id=645432.652389>>. 5, 22, 34

LI, P.; CUI, B. A comparative study on software vulnerability static analysis techniques and tools. In: IEEE INTERNATIONAL CONFERENCE ON

INFORMATION THEORY AND INFORMATION SECURITY (ICITIS), 2010. **Proceedings...** [S.l.], 2010. p. 521–524. 28

LIU, C.; KUNG, D. C.; HSIA, P.; HSU, C. Object-based data flow testing of web applications. In: ASIA-PACIFIC CONFERENCE ON QUALITY SOFTWARE (APAQS), 1., 2000, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2000. (APAQS '00), p. 7–. ISBN 0-7695-0825-1. Available from: <<http://dl.acm.org/citation.cfm?id=786446.786478>>. 29

LORENZO, A. D.; MEDVET, E.; BARTOLI, A. Automatic string replace by examples. In: ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 15., 2013, Amsterdam, Netherlands. **Proceedings...** New York, NY, USA: ACM, 2013. (GECCO '13), p. 1253–1260. ISBN 978-1-4503-1963-8. Available from: <<http://doi.acm.org/10.1145/2463372.2463532>>. 44

LUCCA, G. A. D.; FASOLINO, A. R.; FARALLI, F.; CARLINI, U. D. Testing web applications. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2002, Montréal, Canada. **Proceedings...** [S.l.], 2002. p. 310–319. ISSN 1063-6773. 29

MARINKE, R. **Geração de testes estruturais para aplicações multithreads: abordagem por statecharts**. 95 p. Master Thesis (Mestrado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011-11-18 2012. Available from: <<http://urlib.net/sid.inpe.br/mtc-m19/2011/11.04.20.08>>. Access in: 01 jun. 2016. 24, 59, 118

MATHUR, A. P. **Foundations of software testing**. Delhi, India: Pearson Education in South Asia, 2008. 689 p. 37

MEMON, A. M.; XIE, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 31, n. 10, p. 884–896, 2005. ISSN 0098-5589. 30

MOORE, M. M. Rule-based detection for reverse engineering user interfaces. In: WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), 3., 1996, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1996. (WCRE '96), p. 42. ISBN 0-8186-7674-4. Available from: <<http://dl.acm.org/citation.cfm?id=525595.836966>>. 12

MORI, G.; PATERNO, F.; SANTORO, C. CTTE: support for developing and analyzing task models for interactive system design. **IEEE Trans. Software Eng.**, p. 797–813, 2002. 12

MURPHY, C.; SHEN, K.; KAISER, G. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING VERIFICATION AND VALIDATION, 2009, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2009. (ICST '09), p. 436–445. ISBN 978-0-7695-3601-9. Available from: <<http://dx.doi.org/10.1109/ICST.2009.19>>. 30

MYERS, G. **The art of software testing**. 3. ed. [S.l.]: John Wiley & Sons, 2011. 1, 10, 18, 20, 21

NARDI, P. A. **On test oracles for Simulink-like models**. 2013. Tese (Doutorado em Ciências de Computação e Matemática Computacional) — Universidade de São Paulo - USP, São Carlos, 2014. Available from: <<http://www.teses.usp.br/teses/disponiveis/55/55134/tde-26032014-104734/>>. 30

NETBEANS. **NetBeans IDE**. 2015. Available from: <<https://netbeans.org/>>. Access in: 18 mar 2015. 125

PETERS, D. K.; PARNAS, D. L. Using test oracles generated from program documentation. **IEEE Transactions on Software Engineering**, v. 24, n. 3, p. 161–173, Mar 1998. ISSN 0098-5589. 31

PETERSON, J. L. **Petri net theory and modeling of systems**. London: Prentice Hall International, 1981. 26

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. [S.l.]: McGraw-Hill International Editions, 2014. 8th edition. 1, 2, 9, 18, 23

REDMONK. **The RedMonk programming language rankings**. 2016. Available from: <<https://redmonk.com/sogrady/category/programming-languages/>>. Access in: 2 may 2016. 6

SANGWAN, O. P.; BHATIA, P. K.; SINGH, Y. Radial basis function neural network based approach to test oracle. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 36, n. 5, p. 1–5, sep. 2011. ISSN 0163-5948. Available from: <<http://doi.acm.org/10.1145/2020976.2020992>>. 29

SANTIAGO JÚNIOR, V. A. **SOLIMVA: a methodology for generating model-based test cases from natural language requirements and**

**detecting incompleteness in software specifications.** 264 p. PhD Thesis (PhD) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011-12-12 2011. Available from: <<http://urlib.net/sid.inpe.br/mtc-m19/2011/11.07.23.30>>. Access in: 01 jun. 2016. 2, 3, 4, 25, 31

SANTIAGO, V.; SILVA, W. P.; VIJAYKUMAR, N. L. Shortening test case execution time for embedded software. In: INTERNATIONAL CONFERENCE ON SECURE SYSTEM INTEGRATION AND RELIABILITY IMPROVEMENT (SSIRI), 2., 2008. **Proceedings...** [S.l.], 2008. p. 81 –88. 3

SANTIAGO, V.; VIJAYKUMAR, N. L.; GUIMARÃES, D.; AMARAL, A. S.; FERREIRA, E. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING VERIFICATION AND VALIDATION (ICST 2008), 1., 2008, Lillehammer, Noruega. **Proceedings...** Lillehammer, Noruega, 2008. 30

SCHMEELK, S. Open source programming assistance for students. In: INTERNATIONAL CONFERENCE ON EDUCATIONAL AND INFORMATION TECHNOLOGY (ICEIT), 2010. **Proceedings...** [S.l.], 2010. v. 1, p. V1–538–V1–545. 29

SCITOOLS. **Understand**<sup>®</sup>: understand your code. 2015. Available from: <<https://scitools.com>>. Access in: 18 mar 2015. 15, 111

SHEN, H.; FANG, J.; ZHAO, J. EFindBugs: effective error ranking for FindBugs. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST), 4., 2011. **Proceedings...** [S.l.], 2011. p. 299–308. 29

SINGH, A.; KANG, S.; BAJWA, S. Metamorphic testing: using the properties of SUT. **International Journal of Computer Technology and Applications**, v. 2, n. 5, p. 1334–1336, 2011. 30

SKIENA, S. S. **The algorithm design manual.** 2. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008. ISBN 1848000693, 9781848000698. 116, 117

SOMMERVILLE, I. **Software engineering.** 9. ed. [S.l.]: Addison Wesley, 2010. 568 p. 1, 2, 10, 13, 16, 17, 23, 24, 71



SONARQUBE. **SonarQube™ java analyzer**: the only rule engine you need. 2015. Available from: <<http://www.sonarqube.org>>. Access in: 18 mar 2015. 15, 111

SUN, C.; WANG, G.; MU, B.; LIU, H.; WANG, Z.; CHEN, T. Y. Metamorphic testing for web services: framework and a case study. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES (ICWS), 2011, Washington, DC. **Proceedings...** [S.l.], 2011. p. 283 –290. 30

TIOBE. **TIOBE index for april 2016**. 2016. Available from: <[http://www.tiobe.com/tiobe\\_index/](http://www.tiobe.com/tiobe_index/)>. Access in: 2 may 2016. 6

TONELLA, P.; RICCA, F. A 2-layer model for the white-box testing of web applications. In: IEEE INTERNATIONAL WORKSHOP ON WEB SITE EVOLUTION (WSE), 6., 2004, Chicago, IL, USA. **Proceedings...** IEEE, 2004. p. 11–19. ISSN 1550-4441. Available from: <<http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9691>>. 29

TORRI, L.; FACHINI, G.; STEINFELD, L.; CAMARA, V.; CARRO, L.; COTA, E. An evaluation of free/open source static analysis tools applied to embedded software. In: LATIN AMERICAN TEST WORKSHOP (LATW), 11., 2010. **Proceedings...** [S.l.], 2010. p. 1–6. 28

TU, D.; CHEN, R.; DU, Z.; LIU, Y. A method of log file analysis for test oracle. In: INTERNATIONAL CONFERENCE ON SCALABLE COMPUTING AND COMMUNICATIONS; EIGHTH INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTING, 2009, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2009. (SCALCOM-EMBEDDED COM '09), p. 351–354. ISBN 978-0-7695-3825-9. Available from: <<http://dx.doi.org/10.1109/EmbeddedCom-ScalCom.2009.69>>. 30

UBM TECH ELECTRONICS. **2014 embedded market study - then, now: what's next?** 2014. Available from: <<http://bd.eduweb.hhs.nl/es/2014-embedded-market-study-then-now-whatsnext>>. Access in: 7 apr 2016. 6

UML. **Documents associated with unified modeling language (UML)**. 2015. Available from: <<http://www.omg.org/spec/UML/2.5/>>. Access in: 03 apr. 2015. 25, 52

WANG, F.; YAO, L.; WU, J. Intelligent test oracle construction for reactive systems without explicit specifications. In: INTERNATIONAL CONFERENCE ON DEPENDABLE, AUTONOMIC AND SECURE COMPUTING (DASC), 9., 2011. **Proceedings...** [S.l.], 2011. p. 89–96. 31

WARD, P. T.; MELLOR, S. J. **Structured development for real-time systems**: essential modeling techniques. [S.l.]: Prentice Hall, 1985. 26

WENDEHALS, L. Improving design pattern instance recognition by dynamic analysis. In: WORKSHOP ON DYNAMIC ANALYSIS (WODA), 2003, Portland, USA. **Proceedings...** [S.l.], 2003. 19

XIAO, H.; ZHIYI, M.; WEIZHONG, S.; SHAO, G. A metamodel for the notation of graphical modeling languages. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 31., 2007, Beijing. **Proceedings...** IEEE, 2007. v. 1, p. 219–224. Available from: <<http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4290962>>. 25

XIE, Q.; MEMON, A. M. Designing and comparing automated test oracles for GUI-based software applications. **ACM Transactions on Software Engineering and Methodology**, v. 16, n. 1, p. 4, 2007. 26

## Appendix A

### A.1 Additional Information About Regular Expressions

Tables [A.1](#) and [A.2](#) shows all regular expressions which complement the contents of Tables [3.5](#) and [3.6](#).

Table A.1 - Full code line *regex* used in REACTOR.

Description	Regex	Recognized Standard
Primitive type declaration	<code>^[ ](boolean char byte short int) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$</code>	<code>int var ;</code>
Object declaration	<code>^[ ] [A-Z] [a-zA-Z0-9_]* [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$</code>	<code>String var ;</code>
String assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [^\\] * [ ] ; [ ] \$</code>	<code>var = "this is a string" ;</code>
Char assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [^ ] {1,1} [ ] ; [ ] \$</code>	<code>var = 'c' ;</code>
Integer assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [ ] [-+] ? [0-9] + ( [eE] [-+] ? [0-9] + ) ? ( F   f   D   d   l   L ) { 0 , 1 } [ ] ; [ ] \$</code>	<code>var = 9 ;</code>
Float assignment (point left)	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [ ] [-+] ? ( \\ . ) [0-9] + ( [eE] [-+] ? [0-9] + ) ? ( F   f   D   d   l   L ) { 0 , 1 } [ ] ; [ ] \$</code>	<code>var = .9 ;</code>
Float assignment (point right)	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [ ] [-+] ? [0-9] + [ ] { 0 , 1 } \\ . ( F   f   D   d   l   L ) { 0 , 1 } [ ] ; [ ] \$</code>	<code>var = 9. ;</code>
Float assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [ ] [-+] ? [0-9] + ( [eE] [-+] ? [0-9] + ) ? ( F   f   D   d   l   L ) { 0 , 1 } [ ] ; [ ] \$</code>	<code>var = 9.9 ;</code>

(Continue)

Table A.1 - Continuation

Description	Regex	Recognized Standard
Boolean assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (true false) [ ] ; [ ] \$$	<code>var = true ;</code>
New object assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (new) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \wedge ( [ ] {0,1} ) ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) {0,1} ) * [ ] {0,1} \wedge [ ] ; [ ] \$$	<code>var = new Object ( ) ;</code>
Array declaration	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ( \wedge [ ] [ ] \wedge [ ] [ ] {0,1} ) + [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	<code>type [ ] arr ;</code>
From array assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ( \wedge [ ] [ ] \wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \wedge [ ] {0,1} ) + [ ] ; [ ] \$$	<code>var = arr [ index ] ;</code>
To array assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ( \wedge [ ] [ ] \wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \wedge [ ] {0,1} ) + [ ] = [ ] [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	<code>arr [ index ] = var ;</code>
New instance of array object	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (new) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ( \wedge [ ] [ ] \wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \wedge [ ] {0,1} ) + [ ] ; [ ] \$$	<code>var = new Object [ ] ;</code>
Variable assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	<code>var = otherVar ;</code>

(Continue)

Table A.1 - Continuation

Description	Regex	Recognized Standard
Unary operation (left signal)	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (- _ \\ + ~ !) [ ] \{0, 1\} [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] ; [ ] \$$	++ i ;
Unary operation (right signal)	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] \{0, 1\} (- _ \\ + ~ !) [ ] ; [ ] \$$	i ++ ;
Operation assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] (@[A-Z]*@) [ ] [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] ; [ ] \$$	var = varA + varB ;
Method calling	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \\ ( [ ] \{0, 1\} ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) \{0, 1\} ) * [ ] \{0, 1\} \\ ) [ ] ; [ ] \$$	method ( parameter ) ;
Method calling from reference	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \{0, 1\} [ . ] [ ] \{0, 1\} [a-zA-Z_][a-zA-Z0-9_]* [ ] \\ ( [ ] \{0, 1\} ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) \{0, 1\} ) * [ ] \{0, 1\} \\ ) [ ] ; [ ] \$$	ref.method ( parameter ) ;
Method calling assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [ ] [a-zA-Z_ - ] [a-zA-Z0-9_]* [ ] \\ ( [ ] \{0, 1\} ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) \{0, 1\} ) * [ ] \{0, 1\} \\ ) [ ] ; [ ] \$$	var = method ( parameter ) ;

(Continue)

Table A.1 - Continuation

Description	Regex	Recognized Standard
Method calling assignment from reference	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] {0,1} ( \\. ) [ ] {0,1} [a-zA-Z_][a-zA-Z0-9_]* [ ] {0,1} \\. ( [ ] {0,1} ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) {0,1} ) * [ ] {0,1} \\. ) [ ] ; [ ] \$</code>	<code>var = ref.method ( parameter ) ;</code>
Type casting	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] \\. ( cast \\. ) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$</code>	<code>var = ( type ) varX;</code>
Print line	<code>^[ ] ( System.out.println   System.out.print   System.err.print   System.err.println ) [ ] \\. ( [ ] {0,1} ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) {0,1} ) * [ ] {0,1} \\. ) [ ] ; [ ] \$</code>	<code>System.out.println ( parameter ) ;</code>
Variable return	<code>^[ ] ( return ) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$</code>	<code>return var ;</code>
Variable return (with parentheses)	<code>^[ ] ( return ) [ ] \\. ( [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \\. ) [ ] ; [ ] \$</code>	<code>return ( var ) ;</code>
Hex number assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] 0[xX][0-9a-fA-F]+ [ ] ; [ ] \$</code>	<code>var = 0x1A ;</code>
Null assignment	<code>^[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] ( null ) [ ] ; [ ] \$</code>	<code>var = null ;</code>

(Continue)

Table A.1 - Continuation

Description	Regex	Recognized Standard
Collection declaration	$\wedge [ ] \{0,1\} [a-zA-Z_][a-zA-Z0-9_]* [ ] < [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ( \wedge [ ] \wedge [ ] ) * ( [ ] \{0,1\} , [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \{0,1\} ( \wedge [ ] \wedge [ ] ) * ) [ ] \{0,1\} > [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	Class < type > var ;
New collection assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (new) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] < [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ( \wedge [ ] \wedge [ ] ) * ( [ ] \{0,1\} , [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \{0,1\} ( \wedge [ ] \wedge [ ] ) * ) * [ ] \{0,1\} > [ ] ( [ ] \{0,1\} ( [a-zA-Z_][a-zA-Z0-9_]* ( # ) \{0,1\} ) * [ ] \{0,1\} \wedge [ ] ; [ ] \$$	var = new Class < type > ( ) ;
Throw exception	$\wedge [ ] (throw) [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	throw exception ;
Class attribute assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] [a-zA-Z_][a-zA-Z0-9_]* ( \wedge [ ] \wedge [ ] ) [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	var = class.attribute ;
Reference this assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (this) [ . ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	var = this.ref ;
Reference super assignment	$\wedge [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (super) [ . ] [a-zA-Z_][a-zA-Z0-9_]* [ ] ; [ ] \$$	var = super.ref ;

(Continue)



Table A.1 - Continuation

Description	Regex	Recognized Standard
Java package	<pre> ^((package) [ ]((([a-zA-Z_][a-zA-Z0-9_ ]*)(\.\.)*)+ [ ]{0,1})(;)\$ </pre>	<pre> package reactor.CodeLineMatcher ; </pre>
Java import	<pre> ^((import) [ ]((([a-zA-Z_][a-zA-Z0-9_ ]*)(\.\.)*)+ [ ]{0,1})(;)\$ </pre>	<pre> import java.util.regex.Matcher ; </pre>
Class declaration	<pre> ^((public  private  protected )){0,1}(final ){0,1}(abstract )){0,1}(class  interface )[a-zA-Z_][a-zA-Z0-9_]* [ ]((extends) [ ][a-zA-Z_][a-zA-Z0-9_]* [ ]{0,1})* * [ ]{0,1}((implements) [ ]([a-zA-Z0-9_\\ &lt; &gt; ])* [ ]{0,1}[ , ]*[ ]{0,1})* * [ ]{0,1}((\{)\$ </pre>	<pre> public class CodeMatcher extends Matcher { </pre>
Method declaration	<pre> ^((public  private  protected ){0,1}(static )){0,1}(final ){0,1}(abstract )){0,1}(synchronized ){0,1}([a-zA-Z_][a-zA-Z0-9_]* [ ]{0,1}[\ &lt; &gt; \  [ ]{0,1} [ ]{0,1}[a-zA-Z0-9_ ]* [ ]{0,1}[\ &lt; &gt; \  [ ]{0,1}]{0,1}]{0,1} [ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]* [ ]{0,1} \ ( [ ]* ([a-zA-Z_][a-zA-Z0-9_ \  [ ]{0,1} \  &lt; \  &gt; ])* [ ][a-zA-Z_][a-zA-Z0-9_ \  [ ]{0,1} \  &lt; \  &gt; , ])* * \  \ ( [ ]{0,1} ) (throws [a-zA-Z0-9, ])* * ( \  \ { ) \$ </pre>	<pre> public void execute() throws IOException { </pre>

(Continue)



Table A.1 - Continuation

Description	Regex	Recognized Standard
Control structure <i>if</i>	<code>^(if)[ ]\([a-zA-Z0-9 &gt; &lt; \[ \] }\{+ - * \/ \/% \\!; ; . , \/)\([* ]\)\{ } [ ] \$</code>	<code>if ( varA &gt;= varB ) {</code>
Short <i>if</i>	<code>^[a-zA-Z0-9 \\!%*( )\ -+ \{ \} \[ \] &lt; &gt; .   = : ; \\]* [ ] ( ?) [ ] [a-zA-Z0-9 \\!%*( )\ -+ \{ \} \[ \] &lt; &gt; . = /]* [ ] ( :  [ ]  [a-zA-Z0-9 \\!%*( )\ -+ \{ \} \[ \] &lt; &gt; . = /]* [ ] [ ] ( ; ) [ ] \$</code>	<code>type var = ( condition ) ? true : false ;</code>
Control structure <i>else-if</i>	<code>^[ ] (else if) [ ] \([a-zA-Z0-9 &gt; &lt; \[ \] }\{+ - * \/ \/% \\!; ; . , \/)\([* ] [ ] \{ } [ ] \$</code>	<code>} else if ( varA &lt; varB ) {</code>
Control structure <i>else</i>	<code>^[ ] (else) [ ] \{ } [ ] \$</code>	<code>} else {</code>
Control structure <i>for</i>	<code>^(for)[ ]\([a-zA-Z0-9 &gt; &lt; \[ \] }\{+ - * \/ \/% \\!; ; . , \/)\([* ]\)\{ } [ ] \$</code>	<code>for ( int i = 0 ; i &lt;= var ; i++ ) {</code>
Control structure compact <i>for</i>	<code>^(for)[ ]\([a-zA-Z0-9 &gt; &lt; \[ \] }\{+ - * \/ \/% \\!; ; , \/.\) \([*(:)[a-zA-Z0-9 &gt; &lt; \[ \] }  \{+ - * \/ \/% !; , \/.\) \([* ]\)\{ } [ ] \$</code>	<code>for ( var : ref.getVars ( ) ) {</code>
Control structure <i>while</i>	<code>^(while)[ ]\([a-zA-Z0-9 &gt; &lt; \[ \] }\{+ - * \/ \/% \\!; ; , \/)\([* ]\)\{ } [ ] \$</code>	<code>while ( varA &gt;= varB ) {</code>
Control structure <i>do</i>	<code>^(do)[ ]\{ } [ ] \$</code>	<code>do {</code>

(Continue)

Table A.1 - Continuation

Description	Regex	Recognized Standard
Control structure (do-while)	<code>^}[ ] (while)[ ] \ ([a-zA-Z0-9]   &gt;   &lt;   [ ]   }   {   +   -   *   /   %   !   :   ;   \   /   \   ) \ ( [ ] * [ ]   ) [ ] \ ) [ ] \$</code>	<code>} while ( var &gt; 0 ) ;</code>
Control structure <i>switch</i>	<code>^(switch)[ ] \ ( [ ] [a-zA-Z0-9]* [ ]   ) [ ] \ { [ ] [ ] \$</code>	<code>switch ( var ) {</code>
Control structure <i>switch-case</i>	<code>^(case)[ ] ( ) [ ] \$</code>	<code>case :</code>
Control structure <i>switch-break</i>	<code>^(break)[ ] ( ) [ ] \$</code>	<code>break ;</code>
Control structure <i>switch-continue</i>	<code>^(continue)[ ] ( ) [ ] \$</code>	<code>continue ;</code>
Control structure <i>switch-default</i>	<code>^(default)[ ] ( ) [ ] \$</code>	<code>default :</code>
Control structure <i>try</i>	<code>^(try)[ ] \ { [ ] [ ] \$</code>	<code>try {</code>
Control structure <i>try-catch</i>	<code>^}[ ] (catch)[ ] \ ( [ ] [a-zA-Z0-9]* [ ]   ) [ ] \ { [ ] [ ] \$</code>	<code>} catch ( Exception e ) {</code>
Control structure <i>try-finally</i>	<code>^}[ ] (finally)[ ] \ { [ ] [ ] \$</code>	<code>} finally {</code>
Scope closing	<code>^}[ ] \$</code>	<code>}</code>

Table A.2 - Inline code *regex* used in REACTOR.

Description	Regex	Recognized Standard
Variable declaration	<pre>[ ] [a-zA-Z_][a-zA-Z0-9_]*([ ]{0,1})\ \ [ ]\ \ [ ] ]{0,1})*[ ] [a-zA-Z_][a-zA-Z0-9_]*[ ]{0,1}[\ \ ; =]</pre>	type variable
Method calling assignment from reference	<pre>[ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]*[ ]=[ ] [a-zA-Z_- ][a-zA-Z0-9_]*((\ \.) [a-zA-Z_][a-zA-Z0-9_]*[ ]\ \ [ ]{0,1}){([a-zA-Z_][a-zA-Z0-9_]* ( # )}{0,1})* [ ]{0,1}\ \ ) [ ]{0,1}</pre>	var = ref.method ( parameter )
Method calling assignment	<pre>[ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]*[ ]=[ ] [a-zA-Z_- ][a-zA-Z0-9_]*[ ]\ \ [ ]{0,1}([a-zA-Z_- ][a-zA-Z0-9_]* ( # )){0,1})*[ ]{0,1}\ \ ) [ ]{0,1}</pre>	var = method ( parameter )
Method calling from reference	<pre>[ ]{0,1}[ ] [a-zA-Z_][a-zA-Z0-9_]*((\ \.) [a-zA-Z_- ][a-zA-Z0-9_]*[ ]{0,1})\ \ [ ]{0,1})([a-zA-Z_- ][a-zA-Z0-9_]* ( # )){0,1})*[ ]{0,1}\ \ ) [ ]{0,1}</pre>	ref.method ( parameter )
Method calling	<pre>[ \ \ &lt;(\ ); \ \ #\ \ = \ \ + \ \  - \ \  * \ \ &amp;\ \  %\ \  \$\ \  \@&lt;(return)&gt; ] [ a-zA-Z_][a-zA-Z0-9_]*[ ] [ ]\ \ [ ]{0,1})([a-zA-Z_- ][a-zA-Z0-9_]* ( # )){0,1})*[ ]{0,1}\ \ ) [ ]{0,1}</pre>	method ( parameter )
Collection declaration	<pre>[ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]*[ ] [ ]&lt; [ ] [a-zA-Z_- ][a-zA-Z0-9_]*[ ] ( \ \ [ ] \ \  )*( [ ]{0,1} , [ ] [a-zA-Z_- ][a-zA-Z0-9_]*[ ] {0,1} ) ( \ \ [ ] \ \  )*( [ ]{0,1} )&gt; [ ][a-zA-Z_][a-zA-Z0-9_]*[ ] {0,1} [\ \ =]</pre>	Class < type > var

(Continue)

Table A.2 - Continuation

Description	Regex	Recognized Standard
New object assignment	<pre>[ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]*[ ]=[ ](new)[ ][a-zA-Z_][a-zA-Z0-9_]*[ ]\(\([ ]{0,1}\)([a-zA-Z_- ][a-zA-Z0-9_]*(# )\{0,1\})*\[ ]{0,1}\)\)\[ ]</pre>	var = new Object ( )
New object	<pre>[ ](new)[ ] [a-zA-Z_][a-zA-Z0-9_]*[ ]\(\([ ]{0,1}\)([a-zA-Z_][a-zA-Z0-9_]*(# )\{0,1\})*\[ ]{0,1}\)\)\[ ]</pre>	new Object ( )
New collection assignment	<pre>[ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]*[ ]=[ ](new)[ ][a-zA-Z_][a-zA-Z0-9_]*[ ]&lt;[ ] [a-zA-Z_- ][a-zA-Z0-9_]*[ ](\([ ]{0,1}\))*(\([ ]{0,1}, [ ] [a-zA-Z_- ][a-zA-Z0-9_]*[ ]{0,1}\)(\([ ]{0,1}\))*[ ]{0,1}\)&gt;[ ]\(\([ ]{0,1}\)([a-zA-Z_][a-zA-Z0-9_]*(# )\{0,1\})*\[ ]{0,1}\)\)\[ ]{0,1}</pre>	var = new Class < type > ( )
New collection	<pre>[ ](new)[ ] [a-zA-Z_][a-zA-Z0-9_]*[ ]&lt;[ ] [a-zA-Z_- ][a-zA-Z0-9_]*[ ]&gt;[ ]\(\([ ]{0,1}\)([a-zA-Z_- ][a-zA-Z0-9_]*(# )\{0,1\})*\[ ]{0,1}\)\)\[ ]</pre>	new Class < type > ( )
New array assignment	<pre>[ ]{0,1}[a-zA-Z_][a-zA-Z0-9_]*[ ]=[ ](new)[ ]+ dt + [ ](\([ ] [a-zA-Z_][a-zA-Z0-9_]*[ ]\)\)\[ ]{0,1}\)+[ ]</pre>	var = new type [ ]
New array	<pre>[ ]{0,1}(new)[ ] [a-zA-Z_][a-zA-Z0-9_]*[ ](\([ ] [ ][a-zA-Z_][a-zA-Z0-9_]*[ ]\)\)\[ ]{0,1}\)+[ ]</pre>	new type [ ]

(Continue)



Table A.2 - Continuation

Description	Regex	Recognized Standard
Boolean value assignment	[ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] = [ ] (true false) [ ]	var = true
Boolean value	[ ] (true false) [ ]	true
Type casting	[ ] {0,1} \\\ ( [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ] \\\ ) [ ] [ ] [a-zA-Z_][a-zA-Z0-9_]* [ ]	( type ) var
Public attribute variable	[ ] [a-zA-Z_][a-zA-Z0-9_]* (\\ .) [a-zA-Z_-] [ ] [a-zA-Z0-9_]* [ ] {0,1} [ ] [ \\ # @ \\ \\ . \\ ; \\ < \\ ( \\ \\ > \\ ) \\ \\ > ]	ref.variable
Reference <i>this</i>	[ ] (this) [ ] * (\\ .) [ ] *	this.
Reference <i>super</i>	[ ] (super) [ ] * (\\ .) [ ] *	super.



## A.2 Additional Information About Source Code Classification

Table A.3 shows the set of standard lines for source code classification.

Table A.3 - Set of standard lines for source code classification.

1	type v;	27	} else {
2	v = "String";	28	v = ( ) ? 1 : 0 ;
3	v = 'c';	29	for ( ; ; ) {
4	v = 9.9;	30	for ( : ) {
5	v = boolean;	31	while ( ) {
6	v = new Class();	32	do {
7	type[] v;	33	} while ( ) ;
8	v = x[i];	34	switch ( ) {
9	x[i] = v;	35	case :
10	x = new type[i];	36	break ;
11	v = x;	37	continue ;
12	v = op x;	38	default :
13	v = x op;	39	try {
14	v = x op y;	40	} catch ( ) {
15	method(x);	41	} finally {
16	v.method(x);	42	}
17	v = method(x);	43	throw exception ;
18	v = x.method(y);	44	class.attribute ;
19	v = (type) x;	45	package pack ;
20	return v;	46	import class ;
21	v = 0x00;	47	class Class {
22	v = null;	48	Method ( ) {
23	Collection < Class > v;	49	Constructor ( ) {
24	v = new Collection < Class > ();	50	print ( v ) ;
25	if ( ) {	51	super ;
26	} else if ( ) {		



## Appendix B

### B.1 Additional Information About Evaluation of Hanoi Towers

Source Code B.1 and B.2 shows the source code used in evaluation of this case study.

Code B.1 - Hanoi Towers source code (Hanoi class)

---

```
1  /*
2  * To change this template, choose Tools / Templates
3  * and open the template in the editor.
4  */
5  package iut.TestHanoi;
6
7  /**
8   *
9   * @author Aarantes
10  */
11 public class Hanoi {
12
13     int movimientos;
14
15     public Hanoi() {
16         movimientos = 0;
17     }
18
19     public int getMovimientos() {
20         return movimientos;
21     }
22
23     public void execute(int n, int O, int D, int T) {
24         //System.out.println("execute(" + n + " , " + O + " , " + D + "
25         , " + T + " );");
26         if (n > 0) {
27             execute(n - 1, O, T, D);
28             movimientos++;
29             execute(n - 1, T, D, O);
30         }
31     }
}
```

---

Code B.2 - Hanoi Towers source code (Main class)

---

```
1  /*
2  * To change this template, choose Tools / Templates
3  * and open the template in the editor.
4  */
5  package iut.TestHanoi;
6
7  /**
8   *
9   * @author Aarantes
10  */
11  public class Main {
12
13      public static void main(String [] args) {
14          int n = 4;
15          Hanoi hanoi = new Hanoi();
16          hanoi.execute(n, 1, 3, 2);
17          int m = hanoi.getMovimentos();
18          System.out.println(m);
19      }
20  }
```

---

## B.2 Additional Information About Evaluation of Triangle Classification

Source Code B.3 and B.4 shows the source code used in evaluation of this case study.

Code B.3 - Triangle Classification source code (Triangle class)

---

```
1  /*
2  * To change this template, choose Tools / Templates
3  * and open the template in the editor.
4  */
5  package iut.TestTriangle;
6
7  /**
8   *
9   * @author Alessandro
10  */
11  public class Triangle extends Main {
12
13      public String Equ = "Equilatero";
14      public String Iso = "Isoceles";
15      public String Esc = "Escaleno";
16
17      public String ChecarTriangulo(double a, double b, double c) {
18          String res = null;
19
20          if (a == b && a == c && b == c) {
21              res = Equ;
22          } else if (a == b || a == c || b == c) {
23              res = Iso;
24          } else if (a != b && a != c && b != c) {
25              res = Esc;
26          }
27
28          return res;
29      }
30  }
```

---

Code B.4 - Triangle Classification source code (Main class)

---

```
1  /*
2  * To change this template, choose Tools / Templates
3  * and open the template in the editor.
4  */
5  package iut.TestTriangle;
6
7  import java.util.ArrayList;
8
9  /**
10 *
11 * @author Alessandro
12 */
13 public class Main {
14
15     public static void main(String [] args) {
16
17         Triangle t = new Triangle();
18
19         double input_a = 0;
20         double input_b = 0;
21         double input_c = 0;
22
23         String type = t.ChecarTriangulo(input_a, input_b, input_c);
24
25         System.out.println(type);
26     }
27 }
```

---

### B.3 Additional Information About Evaluation of Quick Sort

Source Code B.5 and B.6 shows the source code used in evaluation of this case study.

Code B.5 - Quick Sort source code (Quicksort class)

---

```
1  /*
2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5  package iut.TestQuicksort;
6
7  public class Quicksort {
8
9      private int [] numbers;
10     private int number;
11
12     public void sort(int [] values) {
13         // Check for empty or null array
14         //     if (values == null || values.length == 0) {
15         //         return;
16         //     }
17         this.numbers = values;
18         number = values.length;
19         quicksrt(0, number - 1);
20     }
21
22     private void quicksrt(int low, int high) {
23         int i = low, j = high;
24         // Get the pivot element from the middle of the list
25         int pivot = numbers[low + (high - low) / 2];
26
27         // Divide into two lists
28         while (i <= j) {
29             // If the current value from the left list is smaller than
30             // the pivot
31             // element then get the next element from the left list
32             while (numbers[i] < pivot) {
33                 i++;
34             }
35             // If the current value from the right list is larger than
36             // the pivot
37             // element then get the next element from the right list
38             while (numbers[j] > pivot) {
39                 j--;
40             }
41         }
42     }
43 }
```

```

39
40     // If we have found a values in the left list which is
41     // the pivot element and if we have found a value in the
42     // which is smaller than the pivot element then we exchange
43     // values.
44     // As we are done we can increase i and j
45     if (i <= j) {
46         exchange(i, j);
47         i++;
48         j--;
49     }
50 }
51 // Recursion
52 if (low < j) {
53     quicksrt(low, j);
54 }
55 if (i < high) {
56     quicksrt(i, high);
57 }
58 }
59
60 private void exchange(int i, int j) {
61     int temp = numbers[i];
62     numbers[i] = numbers[j];
63     numbers[j] = temp;
64 }
65 }

```

---



Code B.6 - Quick Sort source code (Main class)

---

```
1  /*
2  * To change this template, choose Tools / Templates
3  * and open the template in the editor.
4  */
5  package iut.TestQuicksort;
6
7  import iut.TestQuicksort.Quicksort;
8
9  /**
10 *
11 * @author Aarantes
12 */
13 public class Main {
14
15     public static void main(String[] args) {
16
17         int n = 3;
18         int unsorted = 0;
19
20         int[] values = new int[n];
21
22         for (int i = 0; i < n; i++) {
23             values[i] = 1 + (int) (Math.random() * 100);
24         }
25
26         System.out.println("Unsorted!");
27         for (int i = 0; i < values.length; i++) {
28             if (i > 0) {
29                 if (values[i - 1] > values[i]) {
30                     unsorted++;
31                 }
32             }
33             System.out.println(values[i]);
34         }
35
36         Quicksort m = new Quicksort();
37         m.sort(values);
38
39         System.out.println("Unsorted numbers: " + unsorted);
40
41         unsorted = 0;
42
43         System.out.println("");
44
```

```
45     System.out.println("Sorted!");
46     for (int i = 0; i < values.length; i++) {
47         if (i > 0) {
48             if (values[i - 1] > values[i]) {
49                 unsorted++;
50             }
51         }
52         System.out.println(values[i]);
53     }
54     System.out.println("Unsorted numbers: " + unsorted);
55 }
56 }
```

---

## B.4 Additional Information About Evaluation of Merge Sort

Source Code B.7 and B.8 shows the source code used in evaluation of this case study.

Code B.7 - Merge Sort source code (Mergesort class)

---

```
1 package iut.TestMergesort;
2
3 public class Mergesort {
4     private int [] numbers;
5     private int [] helper;
6
7     private int number;
8
9     public void sort(int [] values) {
10        this.numbers = values;
11        number = values.length;
12        this.helper = new int[number];
13        mergesort(0, number - 1);
14    }
15
16    private void mergesort(int low, int high) {
17        // Check if low is smaller then high, if not then the
18        // array is sorted
19        if (low < high) {
20            // Get the index of the element which is in the
21            // middle
22            int middle = (low + high) / 2;
23            // Sort the left side of the array
24            mergesort(low, middle);
25            // Sort the right side of the array
26            mergesort(middle + 1, high);
27            // Combine them both
28            merge(low, middle, high);
29        }
30    }
31
32    private void merge(int low, int middle, int high) {
33        // Copy both parts into the helper array
34        for (int i = low; i <= high; i++) {
35            helper[i] = numbers[i];
36        }
37
38        int i = low;
39        int j = middle + 1;
```

```

39     int k = low;
40     // Copy the smallest values from either the left or the
      // right side back
41     // to the original array
42     while (i <= middle && j <= high) {
43         if (helper[i] <= helper[j]) {
44             numbers[k] = helper[i];
45             i++;
46         } else {
47             numbers[k] = helper[j];
48             j++;
49         }
50         k++;
51     }
52     // Copy the rest of the left side of the array into the
      // target array
53     while (i <= middle) {
54         numbers[k] = helper[i];
55         k++;
56         i++;
57     }
58
59     }
60 }

```

---

Code B.8 - Merge Sort source code (Main class)

---

```
1  /*
2  * To change this template, choose Tools / Templates
3  * and open the template in the editor.
4  */
5  package iut.TestMergesort;
6
7  /**
8   *
9   * @author Aarantes
10  */
11  public class Main {
12
13      public static void main(String [] args) {
14
15          int numbers = 30;
16          int unsorted = 0;
17
18          int [] values = new int [numbers];
19
20          for (int i = 0; i < numbers; i++) {
21              values[i] = 1 + (int) (Math.random() * 100);
22          }
23
24          System.out.println("Unsorted!");
25          for (int i = 0; i < values.length; i++) {
26              if (i > 0) {
27                  if (values[i - 1] > values[i]) {
28                      unsorted++;
29                  }
30              }
31              System.out.println(values[i]);
32          }
33
34          Mergesort m = new Mergesort();
35          m.sort(values);
36
37          System.out.println("Unsorted numbers: " + unsorted);
38
39          unsorted = 0;
40
41          System.out.println("");
42
43          System.out.println("Sorted!");
44          for (int i = 0; i < values.length; i++) {
```

```
45         if (i > 0) {
46             if (values[i - 1] > values[i]) {
47                 unsorted++;
48             }
49         }
50         System.out.println(values[i]);
51     }
52     System.out.println("Unsorted numbers: " + unsorted);
53 }
54 }
```

---

## B.5 Additional Information About Evaluation of Bubble Sort

Source Code B.9 and B.10 shows the source code used in evaluation of this case study.

Code B.9 - Bubble Sort source code (Bubblesort class)

---

```
1  /*
2  * To change this license header, choose License Headers in Project
3  * Properties.
4  * To change this template file, choose Tools | Templates
5  * and open the template in the editor.
6  */
7
8  package iut.TestBubblesort;
9
10
11
12
13
14  class Bubblesort {
15
16      public void sort(int [] array) {
17          int n, c, d, swap;
18          n = array.length;
19
20          for (c = 0; c < (n - 1); c++) {
21              for (d = 0; d < n - c - 1; d++) {
22                  if (array[d] > array[d + 1]) /* For descending order
23                  use < */ {
24                      swap = array[d];
25                      array[d] = array[d + 1];
26                      array[d + 1] = swap;
27                  }
28              }
29          }
30      }
31      // public static void main(String [] args) {
32      //     int n, c, d, swap;
33      //     Scanner in = new Scanner(System.in);
34      //
35      //     System.out.println("Input number of integers to sort");
36      //     n = in.nextInt();
37      //
```

```

38 //      int array[] = new int[n];
39 //
40 //      System.out.println("Enter " + n + " integers");
41 //
42 //      for (c = 0; c < n; c++) {
43 //          array[c] = in.nextInt();
44 //      }
45 //
46 //      for (c = 0; c < (n - 1); c++) {
47 //          for (d = 0; d < n - c - 1; d++) {
48 //              if (array[d] > array[d + 1]) /* For descending order
         use < */ {
49 //                  swap = array[d];
50 //                  array[d] = array[d + 1];
51 //                  array[d + 1] = swap;
52 //              }
53 //          }
54 //      }
55 //
56 //      System.out.println("Sorted list of numbers");
57 //
58 //      for (c = 0; c < n; c++) {
59 //          System.out.println(array[c]);
60 //      }
61 //  }

```

---



Code B.10 - Bubble Sort source code (Main class)

---

```
1  /*
2  * To change this license header, choose License Headers in Project
3  * Properties.
4  * To change this template file, choose Tools | Templates
5  * and open the template in the editor.
6  */
7
8  package iut.TestBubblesort;
9
10 /**
11 *
12 * @author Aarantes
13 */
14 public class Main {
15
16     public static void main(String [] args) {
17
18         int n = 10;
19         int unsorted = 0;
20
21         int [] values = new int [n];
22
23         for (int i = 0; i < n; i++) {
24             values[i] = 1 + (int) (Math.random() * 100);
25         }
26
27         System.out.println("Unsorted!");
28         for (int i = 0; i < values.length; i++) {
29             if (i > 0) {
30                 if (values[i - 1] > values[i]) {
31                     unsorted++;
32                 }
33             }
34             System.out.println(values[i]);
35         }
36
37         Bubblesort m = new Bubblesort();
38         m.sort(values);
39
40         System.out.println("Unsorted numbers: " + unsorted);
41
42         unsorted = 0;
43
44         System.out.println("");
45     }
46 }
```

```
44     System.out.println("Sorted!");
45     for (int i = 0; i < values.length; i++) {
46         if (i > 0) {
47             if (values[i - 1] > values[i]) {
48                 unsorted++;
49             }
50         }
51         System.out.println(values[i]);
52     }
53     System.out.println("Unsorted numbers: " + unsorted);
54 }
55 }
```

---

## B.6 Additional Information About Evaluation of Insertion Sort

Source Code B.11 and B.12 shows the source code used in evaluation of this case study.

Code B.11 - Insertion Sort source code (Insertionsort class)

---

```
1  /*
2  * To change this license header, choose License Headers in Project
   * Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package iut.InsertionSort;
7
8  import java.io.*;
9
10 class InsertionSort {
11
12     public void sort(long[] arrElements) {
13
14         int max = arrElements.length;
15
16         for (int i = 1; i < max; i++) {
17             int j = i;
18             while (j > 0) {
19                 if (arrElements[j - 1] > arrElements[j]) {
20                     long temp = arrElements[j - 1];
21                     arrElements[j - 1] = arrElements[j];
22                     arrElements[j] = temp;
23                     j--;
24                 } else {
25                     break;
26                 }
27             }
28         }
29     }
30 }
31
32 //     public static void main(String[] args) {
33 //
34 //         String inpstring = "";
35 //         InputStreamReader input = new InputStreamReader(System.in);
36 //         BufferedReader reader = new BufferedReader(input);
37 //
38 //         try {
```

```

39 //          System.out.print("Enter a Number Elements for INSERTION
SORT:");
40 //          inpstring = reader.readLine();
41 //
42 //          long max = Long.parseLong(inpstring);
43 //          long [] arrElements = new long[100];
44 //          for (int i = 0; i < max; i++) {
45 //              System.out.print("Enter [" + (i + 1) + "] Element: ")
;
46 //              inpstring = reader.readLine();
47 //              arrElements[i] = Long.parseLong(inpstring);
48 //          }
49 //
50 //          for (int i = 1; i < max; i++) {
51 //              int j = i;
52 //              while (j > 0) {
53 //                  if (arrElements[j - 1] > arrElements[j]) {
54 //                      long temp = arrElements[j - 1];
55 //                      arrElements[j - 1] = arrElements[j];
56 //                      arrElements[j] = temp;
57 //                      j--;
58 //                  } else {
59 //                      break;
60 //                  }
61 //              }
62 //
63 //              System.out.print("After iteration " + i + ": ");
64 //              for (int k = 0; k < max; k++) {
65 //                  System.out.print(arrElements[k] + " ");
66 //              }
67 //
68 //              System.out.println("/*** " + i + " numbers from the
begining of the array are input and they are sorted ***/");
69 //          }
70 //
71 //          System.out.println("The numbers in ascending orders are
given below:");
72 //          for (int i = 0; i < max; i++) {
73 //              System.out.println(arrElements[i]);
74 //          }
75 //
76 //          } catch (Exception e) {
77 //              e.printStackTrace();
78 //          }
79 //      }

```

---

Code B.12 - Insertion Sort source code (Main class)

---

```
1  /*
2  * To change this license header, choose License Headers in Project
3  * Properties.
4  * To change this template file, choose Tools | Templates
5  * and open the template in the editor.
6  */
7
8  package iut.InsertionSort;
9
10 import iut.TestQuicksort.Quicksort;
11
12 /**
13  *
14  * @author Aarantes
15  */
16 public class Main {
17
18     public static void main(String [] args) {
19
20         int n = 10;
21         int unsorted = 0;
22
23         long [] values = new long [n];
24
25         for (int i = 0; i < n; i++) {
26             values [i] = 1 + (int) (Math.random () * 100);
27         }
28
29         System.out.println ("Unsorted!");
30         for (int i = 0; i < values.length; i++) {
31             if (i > 0) {
32                 if (values [i - 1] > values [i]) {
33                     unsorted++;
34                 }
35             }
36             System.out.println (values [i]);
37         }
38
39         InsertionSort m = new InsertionSort ();
40         m.sort (values);
41
42         System.out.println ("Unsorted numbers: " + unsorted);
43
44         unsorted = 0;
45     }
46 }
```

```
44     System.out.println("");
45
46     System.out.println("Sorted!");
47     for (int i = 0; i < values.length; i++) {
48         if (i > 0) {
49             if (values[i - 1] > values[i]) {
50                 unsorted++;
51             }
52         }
53         System.out.println(values[i]);
54     }
55     System.out.println("Unsorted numbers: " + unsorted);
56 }
57 }
```

---

## B.7 Additional Information About Evaluation of Fibonacci Series

Source Code B.13 shows the source code used in evaluation of this case study.

Code B.13 - Fibonacci Series source code

---

```
1  /*
2  * To change this license header, choose License Headers in Project
   * Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package iut.TestFibonacci;
7
8  /**
9   *
10  * @author Aarantes
11  */
12  public class Fibonacci {
13
14      public static void main(String[] args) {
15          Fibonacci f = new Fibonacci();
16          int n = Integer.MAX_VALUE;
17          int r = 0;
18          r = f.calculate(n);
19      }
20
21      public int calculate(int n) {
22          int proximo = 0, atual = 0, anterior = 1;
23          int r = 0;
24          while (proximo <= n) {
25              proximo = atual + anterior;
26              //System.out.println(atual + " " + anterior);
27              r = proximo;
28              anterior = atual;
29              atual = proximo;
30          }
31          System.out.println(r);
32          return r;
33      }
34  }
```

---

## B.8 Additional Information About Evaluation of Arithmetic Mean

Source Code B.14 and B.16 shows the source code used in evaluation of this case study.

Code B.14 - Arithmetic Mean source code (Average class)

---

```
1  /*
2  * To change this license header, choose License Headers in Project
   Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package iut.TestAverage;
7
8  import java.util.List;
9
10 /**
11  *
12  * @author Aarantes
13  */
14 public class Average {
15
16     private double finalAverage;
17     private boolean condition;
18
19     public double getFinalAverage() {
20         return finalAverage;
21     }
22
23     public void calculateArithmeticAverage(Semester s) {
24         //System.out.println("Iniciando o cÃ¡lculo.");
25         //int tamanho = pValores.size();
26         //for (int i = 0; i < tamanho; i++) {
27         //Semester s = pValores.get(i);
28         double pValor1 = s.getFirstGrade();
29         double pValor2 = s.getSecondGrade();
30         if ((pValor1 >= 0) && (pValor1 <= 10)) {
31             if ((pValor2 >= 0) && (pValor2 <= 10)) {
32                 finalAverage = (pValor1 + pValor2) / 2;
33                 condition = true;
34             } // FIM IF
35         //if (condition) {
36         //    System.out.println("MÃ©dia: " + finalAverage);
37         //} else {
38         //    System.out.println("Informe apenas Notas entre 0 e
```



```
        10!");
39     //}
40     condition = false;
41     } // FIM FOR
42     //System.out.println("Fim do c lculo.");
43     } // FIM METODO
44     //}
45 }
```

---

---

Code B.15 - Arithmetic Mean source code (Main class)

---

```
1  /*
2  * To change this license header, choose License Headers in Project
3  * Properties.
4  * To change this template file, choose Tools | Templates
5  * and open the template in the editor.
6  */
7  package iut.TestAverage;
8
9  /**
10 * @author Aarantes
11 */
12 public class Main {
13
14     public static void main(String [] args) {
15
16         double firstGrade = 2;
17         double secondGrade = 3;
18         Semester s = new Semester();
19         s.setFirstGrade(firstGrade);
20         s.setSecondGrade(secondGrade);
21         Average a = new Average();
22         a.calculateArithmeticAverage(s);
23         double result = 0;
24         result = a.getFinalAverage();
25         System.out.println(result);
26     }
27 }
```

---

```
1  /*
2  * To change this license header, choose License Headers in Project
3  * Properties.
4  * To change this template file, choose Tools | Templates
5  * and open the template in the editor.
6  */
7  package iut.TestAverage;
8
9  /**
10 * @author Aarantes
11 */
12 public class Semester {
13
14     private double firstGrade;
15     private double secondGrade;
16
17     public double getFirstGrade() {
18         return firstGrade;
19     }
20
21     public void setFirstGrade(double firstGrade) {
22         this.firstGrade = firstGrade;
23     }
24
25     public double getSecondGrade() {
26         return secondGrade;
27     }
28
29     public void setSecondGrade(double secondGrade) {
30         this.secondGrade = secondGrade;
31     }
32
33 }
```

---

## B.9 Additional Information About Evaluation of Threads

Source Code B.17, B.18, B.19, B.20, B.21, B.22, B.23 and B.24 shows the source code used in evaluation of this case study.

Code B.17 - Threads source code (Buffer class)

---

```
1 package iut.TestBandera.threads.synchronizedBuffer;
2
3 public class Buffer {
4
5     private int memory = -1;
6     private boolean occupied = true;
7     private int sizeBuffer;
8     private int timeSleep;
9
10    public int getSizeBuffer() {
11        return sizeBuffer;
12    }
13
14    public void setSizeBuffer(int sizeBuffer) {
15        this.sizeBuffer = sizeBuffer;
16    }
17
18    public void setTimeSleep(int timeSleep) {
19        this.timeSleep = timeSleep;
20    }
21
22    public int getTimeSleep() {
23        return timeSleep;
24    }
25
26    public synchronized void writeBuffer(int pValue) {
27        while (!occupied) {
28            try {
29                wait();
30            } catch (InterruptedException e) {
31                e.printStackTrace();
32            }
33        }
34        System.err.println(Thread.currentThread().getName() + "
35        producing: " + pValue);
36        this.memory = pValue;
37        occupied = false;
38        notify();
39    }
```

```
39
40     public synchronized int readBuffer () {
41         while (occupied) {
42             try {
43                 wait ();
44             } catch (InterruptedException e) {
45                 e.printStackTrace ();
46             }
47         }
48         System.err.println (Thread.currentThread ().getName () + "
         consuming: " + this.memory);
49         occupied = true;
50         notify ();
51         return this.memory;
52     }
53
54 }
```

---

Code B.18 - Threads source code (Connector class)

---

```
1 package iut.TestBandera.threads.bandera;
2
3 public class Connector {
4
5     public int queue = -1;
6
7     public synchronized int take() {
8         int value;
9         while (queue < 0) {
10            try {
11                wait();
12            } catch (InterruptedException ex) {
13            }
14        }
15        value = queue;
16        queue = -1;
17        return value;
18    }
19
20    public synchronized void add(int o) {
21        queue = 0;
22        notifyAll();
23    }
24
25    public synchronized void stop() {
26        queue = 0;
27        notifyAll();
28    }
29
30 }
```

---

Code B.19 - Threads source code (Consumer class)

---

```
1 package iut.TestBandera.threads.synchronizedBuffer;
2
3 public class Consumer extends Thread {
4
5     private Buffer buffer;
6
7     public Consumer(Buffer pBuffer) {
8         super("Consumer");
9         buffer = pBuffer;
10    }
11
12    public void run() {
13        int value, total = 0;
14        do {
15            try {
16                Thread.sleep(((int) (Math.random() * buffer.getTimeSleep
17                    (())));
18            } catch (InterruptedException exception) {
19                System.err.println(exception.toString());
20            }
21            value = buffer.readBuffer();
22            total += value;
23        } while (value != buffer.getSizeBuffer());
24        System.err.println(getName() + " finished tasks. Total: " +
25            total);
26    }
27 }
```

---

Code B.20 - Threads source code (Heap class)

---

```
1 package iut.TestBandera.threads.bandera;  
2  
3 public class Heap {  
4     static Connector c1, c2, c3, c4;  
5 }
```

---



Code B.21 - Threads source code (Listener class)

---

```
1 package iut.TestBandera.threads.bandera;
2
3 public class Listener extends Thread {
4
5     public void run() {
6         int tmp = -1;
7         while (tmp != 0) {
8             if ((tmp = Heap.c4.take()) != 0) {
9                 System.out.println("output is " + tmp);
10            }
11        }
12    }
13
14 }
```

---

Code B.22 - Threads source code (Main class)

---

```
1 package iut.TestBandera.threads.synchronizedBuffer;
2
3 import java.io.FileNotFoundException;
4
5 import java.io.IOException;
6
7 public class Main {
8
9     public static void main(String args[]) throws FileNotFoundException
10         , IOException {
11
12         Buffer umBuffer = new Buffer();
13
14         int size = 10;
15         int time = -3000;
16
17         int produced = 0;
18         int consumed = 0;
19
20         umBuffer.setSizeBuffer(size);
21         umBuffer.setTimeSleep(time);
22
23         Producer umProdutor = new Producer(umBuffer);
24         Consumer umConsumidor = new Consumer(umBuffer);
25
26         umProdutor.start();
27         umConsumidor.start();
28
29         produced = umBuffer.getSizeBuffer();
30         consumed = umBuffer.getTimeSleep();
31
32         //System.out.println(produced + " -> " + consumed);
33     }
34 }
```

---

Code B.23 - Threads source code (Producer class)

---

```
1 package iut.TestBandera.threads.synchronizedBuffer;
2
3 public class Producer extends Thread {
4
5     private Buffer buffer;
6
7     public Producer(Buffer pBuffer) {
8         super("Producer");
9         buffer = pBuffer;
10    }
11
12    public void run() {
13        for (int i = 1; i <= buffer.getSizeBuffer(); i++) {
14            try {
15                Thread.sleep((int) (Math.random() * buffer.getTimeSleep
16                    ()));
17            } catch (InterruptedException exception) {
18                System.err.println(exception.toString());
19            }
20            buffer.writeBuffer(i);
21        }
22        System.err.println(getName() + " finished tasks.");
23    }
24 }
```

---

Code B.24 - Threads source code (Stage class)

---

```
1 package iut.TestBandera.threads.bandera;
2
3 public class Stage extends Thread {
4
5     public void run() {
6         int tmp = -1;
7         while (tmp != 0) {
8             if ((tmp = Heap.c1.take()) != 0) {
9                 Heap.c2.add(tmp + 1);
10            }
11        }
12        Heap.c2.stop();
13    }
14
15 }
```

---