On test oracles for Simulink-like models

Paulo Augusto Nardi

On test oracles for Simulink-like models

Paulo Augusto Nardi

Advisor: **Prof. Dr. Márcio Eduardo Delamaro**
Co-advisor: **Prof. Dr. Luciano Baresi**

Doctoral dissertation submitted to the *Instituto de Ciências Matemáticas e de Computação* - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*.

USP – São Carlos
January 2014

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

NN233oo

Nardi, Paulo Augusto
On test oracles for Simulink-like models / Paulo Augusto Nardi; orientador Márcio Eduardo Delamaro; co-orientador Luciano Baresi. -- São Carlos, 2014.
148 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2014.

1. Test oracle. 2. Embedded system. 3. Simulink. 4. Software testing. 5. Software engineering. I. Delamaro, Márcio Eduardo, orient. II. Baresi, Luciano, co-orient. III. Título.

Oráculos de teste para modelos Simulink-*like*

Paulo Augusto Nardi

Orientador: **Prof. Dr. Márcio Eduardo Delamaro**
Co-orientador: **Prof. Dr. Luciano Baresi**

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Janeiro de 2014

Acknowledgments

I would like to thank my advisor, Professor Márcio Eduardo Delamaro, who encouraged and inspired me throughout this course. I also thank Professor Luciano Baresi, who agreed to supervise my visit to Politecnico di Milano, and co-advised this work.

My parents, for the education, creation and existence. My brothers, for the education and friendship.

To Ana Paula, for everything.

My friends from Labes, who made this journey easier. In special: André Endo, Fabiano Ferrari, Lucas Bueno, Marco Graciotto, Rafael Oliveira and Vinicius Durelli. My republic fellow, Douglas Rodrigues, for his support and friendship.

To everyone who welcomed me in Milan. Especially: Leandro Sales Pinto and Lenina, for their friendship, moments of relaxation, tips, tours, lunches, dinners, and technical support. Matteo Rossi, for pointing issues, for the explanations and suggestions. Salvatore and other colleagues from Dipartimento di Elettronica ed Informazione.

Denis and Mauro, for the friendship, hospitality and patience in trying to understand me (io non *uscho*, io esco!). You were my family for nine months. Grazie mille!!!

The examining committee, Profs. Ph.D.'s Ellen Francine Barbosa, Paulo Cesar Masiero, Paulo Henrique Monteiro Borba, Otavio Augusto Lazzarini Lemos and Valdivino Alexandre de Santiago Junior.

Professor Adenilso da Silva Simão, José Carlos Maldonado and Eliane Martins, for the suggestions in the qualifying examination.

Professor Renata Meneghetti, Rodrigo Mello, Rosana Braga and Rudinei Goularte, for the disciplines taught in the course.

The ICMC and its employees, for the support. I thank AGX and Embraer for their cooperation. And to all those who somehow made this journey possible.

Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, process 144626/2009-8) for the financial support throughout the PhD, and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES, process PDEE 6834-10-8) for the financial support of the internship abroad.

Agradecimentos

Quero agradecer, primeiramente, ao meu orientador, prof. Márcio Eduardo Delamaro, que me encorajou e me inspirou ao longo deste curso. Agradeço também ao prof. Luciano Baresi, que aceitou me supervisionar no doutorado sanduíche, na Politecnico di Milano, e co-orientar este trabalho.

A meus pais, pela educação, criação e por existir. A meus irmãos, pela educação e amizade.

À Ana Paula, por tudo.

Aos meus amigos do Labes, que fizeram desta caminhada uma viagem mais leve. Em especial a: André Endo, Fabiano Ferrari, Lucas Bueno, Marco Graciotto, Rafael Oliveira e Vinícius Durelli. Ao companheiro de república, Douglas Rodrigues, pelo apoio e amizade.

A todos que me acolheram em Milão. Especialmente: Leandro Sales Pinto e Lenina, pela amizade, momentos de descontração, dicas, passeios, almoços, jantas, e apoio técnico. A Matteo Rossi, pelas questões levantadas, explicações e sugestões. A Salvatore e demais colegas do Dipartimento di Elettronica ed Informazione.

A Denis e Mauro, pela amizade, hospitalidade e paciência em tentar me entender (io non *uscho*, io esco!). Vocês foram minha família por 9 meses. Grazie mille!!!

À comissão julgadora desta tese, professores doutores Ellen Francine Barbosa, Paulo Cesar Masiero, Paulo Henrique Monteiro Borba, Otavio Augusto Lazzarini Lemos e Valdivino Alexandre de Santiago Junior.

Aos professores Adenilso da Silva Simão, José Carlos Maldonado e Eliane Martins, pelas sugestões e críticas pontuais no exame de qualificação.

Aos professores Renata Meneghetti, Rodrigo Mello, Rosana Braga e Rudinei Goularte, pelas disciplinas ministradas no curso.

Ao ICMC, professores e funcionários, pelo pronto auxílio. Agradeço à AGX e Embraer pela colaboração. E a todos aqueles que de algum modo fizeram possível esta jornada.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, processo 144626/2009-8) pelo apoio financeiro ao longo do doutorado e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES, processo PDEE 6834-10-8) pelo apoio financeiro do estágio no exterior.

No matter where you go, there you are.
-Author unknown

Declaration

I confirm that this dissertation has not been submitted in support of an application for another degree at this or any other teaching or research institution. It is the result of my own work and the use of all material from other sources has been properly and fully acknowledged. Research done in collaboration is also clearly indicated.

Excerpts of this dissertation have been either published or submitted for the appreciation of editorial boards of journals, conferences and workshops, according to the list of publications presented as follows. My contributions to each publication are listed as well.

Journal Papers

- **Nardi, P. A.**; Baresi, L.;Delamaro, M. E.:"On engineering test oracles for Simulink-like models"
  - **Journal:** Software: Practice and Experience
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisors.

Conference and Workshop Papers

- **Nardi, P. A.**; Baresi, L.;Delamaro, M. E.:"Specifying Automated Oracles for Simulink Models"
  - **Event:** 19^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2013)
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisors.

Abstract

Embedded systems are present in many fields of application where failure may be critical. Such systems often possess characteristics that hampers the testing activity, as large amount of produced data and temporal requirements which must be specified and evaluated.

There are tools that support the development of models for analysis and simulation still in the *design* stage. After being evaluated, a model may be used as basis to the implementation. In this case, it is important to ensure that the model is consistent with the specification. Otherwise, a divergence will be propagated to the final code. Therefore, the model must be tested prior to the codification.

Simulink is a standard development and simulation tool for models of embedded systems. Its wide application in the industry has promoted the creation of free-software alternatives, as XCos.

In the literature, there are researches which seek to improve the testing activity for Simulink-like models. The proposed solutions usually focus on test case selection strategies. However, little efforts have been directed to the oracle problem, that is, the difficulty in evaluating if an execution agrees with the specification.

The objective of this doctorate proposal is to provide an oracle generation approach for Simulink-like models which addresses the characteristics previously summarized. Specifically, it is proposed a process, methods, procedures and a tool that enable the partially-automated generation of oracles for such models. As a main contribution, it is expected an improvement in the evaluation process of embedded systems in terms of quality, cost and time.

Resumo

Sistemas embarcados estão presentes em diversas áreas de aplicação em que falhas podem ser críticas. Tais sistemas frequentemente possuem características que tornam a fase de teste particularmente desafiadora, como a produção de grande quantidade de dados e requisitos temporais que precisam ser validados de acordo com a especificação.

Existem ferramentas que auxiliam no desenvolvimento de modelos para análise e simulação do comportamento de sistemas embarcados ainda na fase de *design*. Após ser avaliado, o modelo pode ser usado como base para a implementação. Neste caso, deve-se buscar garantir que um modelo esteja de acordo com a especificação. Do contrário, tal divergência será propagada para a implementação. Portanto, é importante que o modelo seja testado antes da fase de implementação.

Simulink é uma ferramenta-padrão de desenvolvimento e simulação de modelos de sistemas embarcados. Sua ampla aplicação na indústria incentivou a criação de alternativas de software livres como XCos.

Na literatura, existem pesquisas que visam a aprimorar a atividade de teste de modelos Simulink-*like*. As soluções propostas geralmente focam em estratégias de seleção de casos de teste. Mas pouco esforço tem sido direcionado ao problema do oráculo, isto é, na dificuldade em avaliar se a execução está de acordo com a especificação.

O objetivo desta proposta de doutorado é prover uma abordagem de geração de oráculos de teste para modelos simulink-*like* que contemple as características previamente resumidas. Especificamente, é proposto um processo, métodos, procedimentos e uma ferramenta que viabilizem a geração parcialmente automatizada de oráculos de teste para modelos Simulink-*like*. Como contribuição principal, é esperada a melhora da qualidade, custo e tempo do processo de validação de sistemas embarcados suportados por modelagem em Simulink e ferramentas similares.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Objectives and Rationale . . . 3
- 1.3 Contributions . . . 3
- 1.4 Thesis Outline . . . 4

2 General Concepts **5**
- 2.1 Software Testing . . . 5
- 2.2 Test Oracles . . . 6
  - 2.2.1 Oracle Approaches . . . 6
  - 2.2.2 Specification-Based Oracles . . . 7
  - 2.2.3 Metamorphic Relation Based Oracles . . . 12
  - 2.2.4 Machine Learning Based Oracles . . . 13
  - 2.2.5 N-version Based Oracles . . . 14
- 2.3 Simulink . . . 14
- 2.4 Final Remarks . . . 17

3 Evaluating the Application of Test Oracles **19**
- 3.1 Overview of Study Selection . . . 19
- 3.2 Identification of Oracle Categories . . . 20
  - 3.2.1 Specification-Based Oracles . . . 22
  - 3.2.2 Metamorphic Relation Based Oracles . . . 24
  - 3.2.3 Machine Learning Based Oracles . . . 26
  - 3.2.4 N-Version Based Oracles . . . 27
- 3.3 Limitations . . . 27
  - 3.3.1 Limitations of Specification-Based Oracles . . . 28
  - 3.3.2 Limitations of Metamorphic Relation Based Oracles . . . 29
  - 3.3.3 Limitations of Machine Learning Based Oracles . . . 29
  - 3.3.4 Limitations of N-Version Based Oracles . . . 30
- 3.4 Test Oracle Support Tools . . . 30
- 3.5 Quality Criteria Application . . . 33
- 3.6 Threats to Validity . . . 39
- 3.7 Final Remarks . . . 39

4 Designing an Oracle Generator **41**
4.1 Oracle Process . . . 42
4.2 Information Definition . . . 44
4.2.1 Temporal Logic . . . 45
4.2.2 Information Structure . . . 47
4.3 Mapping . . . 51
4.4 Oracle Analysis . . . 53
4.4.1 Off-line Oracle Access . . . 53
4.4.2 On-line Oracle Access . . . 56
4.4.3 Oracle Assumption . . . 56
4.5 Final Remarks . . . 58

5 Automating an Oracle Generator **59**
5.1 Features . . . 60
5.2 Running Example . . . 62
5.2.1 Attribute Definition and Mapping . . . 63
5.2.2 Module Layer . . . 64
5.2.3 OIU Layer . . . 65
5.2.4 Behavior Layer . . . 68
5.2.5 Analysis Execution . . . 69
5.3 Assumptions may Impact the Results . . . 71
5.4 OIU: Trigger-Dependent Rules . . . 73
5.5 Final Remarks . . . 75

6 Apolom: Implementation and Limitations **77**
6.1 The ***bridge*** Package . . . 77
6.2 The ***oracleinformation*** Package . . . 79
6.3 The ***oracleprocedure*** Package . . . 80
6.3.1 Detection of Spurious Occurrences . . . 81
6.4 Limitations . . . 84
6.5 Final Remarks . . . 85

7 Empirical Evaluation **87**
7.1 Issues and Hypothesis Statements . . . 87
7.2 Evaluation 1 . . . 89
7.2.1 Operation . . . 89
7.2.2 Simulation Time of Instrumented Models . . . 91
7.2.3 Language Acceptance . . . 92
7.2.4 Threats to Validity . . . 92
7.3 Evaluation 2 . . . 92
7.3.1 Simulation . . . 93
7.3.2 System Description . . . 93
7.3.3 Common Development Team Concerns . . . 96
7.3.4 Oracle Specification . . . 96
7.3.5 Results . . . 100
7.3.6 Threats to Validity . . . 101
7.4 Evaluation 3 . . . 102

7.4.1 Operation and Results . . . 102
7.4.2 Threats to Validity . . . 108
7.5 Evaluation 4 . . . 108
7.5.1 Operation . . . 108
7.5.2 Results . . . 109
7.5.3 Threats to Validity . . . 112
7.6 Final Remarks . . . 112

8 Other Available Approaches **113**
8.1 Simulink Model Verification . . . 113
8.2 Simulink Verification and Validation . . . 117
8.3 Simulink Design Verifier . . . 117
8.4 REACTIS . . . 119
8.5 T-VEC . . . 121
8.6 Considerations . . . 122

9 Conclusions and Final Considerations **123**
9.1 Contributions . . . 125
9.2 Limitations . . . 125
9.3 Possible Research Directions . . . 126

Bibliography **146**

A Aviation Glossary **147**

List of Figures

2.1 A test oracle scenario (Source: Aichernig (1999)) 8
2.2 A wrapper (Source: Edwards (2001)) 10
2.3 State Machines (Source: Andrews and Zhang (2003)) 11
2.4 Breaking detector. Source: (Chapoutot and Martel, 2009) 16

3.1 Publication by year 20
3.2 Publication by year, by category 21
3.3 Publications by year (Specification-based oracles) 23

4.1 Oracle definition process 42
4.2 A model of a boiler control 49
4.3 Modularization 50
4.4 Modularization of Figure 4.2 51
4.5 Oracle Information layers 51
4.6 Mapping options. Source: (Mathworks, 2013a) 52
4.7 Close interval of occurrences 57
4.8 Undecidable relation between A and B 57

5.1 Rule Wizard 60
5.2 Mapping sample 61
5.3 Instrumented model 62
5.4 Model and folder selection 63
5.5 Intrumentation editor 64
5.6 Module editor 64
5.7 A defined OIU 65
5.8 OIU editor 66
5.9 Behavior selection 66
5.10 Parameter settings 66
5.11 Shortcut selection 67
5.12 Rule generation 67
5.13 Behavior definition 68
5.14 A tooltip 69
5.15 Oracle setup 69

5.16 The configuration . . . 70
5.17 Oracle report . . . 70
5.18 Oracle report: successive assumption . . . 72
5.19 Oracle report: overlapping assumption . . . 72
5.20 Oracle report: constraint failure . . . 73
5.21 Oracle report: critical failure . . . 74
5.22 Oracle report: constraint critical failure . . . 74

6.1 Driver package: parsers and converters . . . 78
6.2 Module structure . . . 79
6.3 OIU structure . . . 79
6.4 Successive assumption analysis . . . 82

7.1 Lorenz Equation . . . 90
7.2 Attribute definition and mapping . . . 90
7.3 Behavior of equation (2) . . . 91
7.4 UAV simulation: waypoints . . . 93
7.5 UAV simplified subsystem diagram . . . 94
7.6 Oracle Report . . . 101
7.7 Relation of performances with different MMS and SMS . . . 104
7.8 Relation of performances when SMS is used . . . 106
7.9 Lorenz Attractor example. Source: (Stewart, 1989) . . . 109
7.10 Lorenz Attractor samples. . . . 110

8.1 Simulink Model Verification . . . 113
8.2 Apolom report . . . 116
8.3 Simulink Design Verifier . . . 118
8.4 A valid proof . . . 119
8.5 A Design Verifier internal error . . . 119
8.6 Reactis stateflow requirement. Source: Reactive Systems (2012) . . . 120
8.7 Reactis stateflow requirement. Source: Reactive Systems (2012) . . . 120

List of Tables

2.1 Simulink blocks 14
2.2 Inport, time and outport 17

3.1 Relation between oracle categories 21
3.2 Specification-based oracles 22
3.3 Last years publications 24
3.4 Oracle Support 30
3.5 Quality criteria application 34

4.1 TRIO Operators 46
4.2 Added functions 46

7.1 Issues and hypothesis statements 87
7.2 Lorenz execution time 103
7.3 Fast Jumper: data interval behind and beyond main memory space 105
7.4 Fast Jumper: large secondary memory space 107
7.5 Fast Jumper: a “real-world” study 107
7.6 Selected mutant operators 108

8.1 SMV blocks 114

Abbreviations and Acronyms

ADT	-	Abstract data type
ANN	-	Artificial neural network
GUI	-	graphical user interface
IFN	-	Info-fuzzy network
JML	-	Java Modeling Language
MITL	-	Microsoft Transformation Language
MMS	-	Main memory space
NN	-	Neural network (as ANN)
OIU	-	Oracle information unit
RBF	-	Radial basis function
ROC	-	Receiver operating characteristic
SDL	-	Specification and Description Language
SDV	-	Simulink Design Verifier
SLA	-	Supervised learning algorithm
SMS	-	Secondary memory space
SMV	-	Simulink Model Verification
SOM	-	Self-organizing map
SUT	-	Software under test
SVM	-	Support vector machine
SVV	-	Simulink Verification and Validation
SWT	-	Standard Widget Toolkit
UAV	-	Unmanned aerial vehicle
VDM	-	Vienna Development Method

Chapter

1

Introduction

Embedded software are the core of many complex systems and their failure may result in death, great economic losses, and severe environmental damages. The key role of these systems, associated with the shortened development cycles within the development organizations and high customer expectations of quality (Lazić and Velašević, 2004), imposes a very careful design and validation to discover potential problems as early as possible.

The design of embedded systems is usually performed through executable models that simulate the behavior of the actual system. Examples of standard system modelers and simulators are Simulink, a commercial tool [1], and free alternatives as XCos [2] and Scicos [3].

However, simulation is often not enough to assess the quality of designed models given that existing defects may not be easy to detect. Moreover, the output data are too large and complex to be analyzed and understood manually. The model quality, which reflects on the system quality, should be assessed through suitable test campaigns (Reicherdt and Glesner, 2012).

In the literature, there are two classical test problems: (i) the reliable test set problem (Chen et al., 2001a) – the definition of a finite test set that if passed would guarantee the correctness of the program, and; (ii) the oracle problem (Weyuker, 1982) – the identification of a procedure to decide whether the obtained result matches the expected result. Usually, the oracle role is played by a human tester.

[1] MathWorks: http://www.mathworks.com/products/simulink/
[2] The Scilab Consortium: http://www.scilab.org/scilab/gallery/xcos
[3] INRIA: http://www.scicos.org/

Whereas it is usually impossible to decide if a program is correct, the purpose of software testing is to determine whether a program contains errors (Goodenough and Gerhart, 1975). Therefore, test oracles are essential components in test harness because it needs to report whether the test results are failures (Chan and Tse, 2013).

Model-driven testing comprises characteristics that hamper the test activity: (i) as mentioned earlier, simulations usually involve large amounts of inputs and outputs which leads to a costly and error-prone comparison process by manual means; (ii) the dynamical nature of simulations commonly demands evaluation of requirements with temporal properties, which is hard to assess manually; and, (iii) similar behaviors may be challenging to evaluate manually.

Considering the infeasibility of generating fully-automated oracles (the oracle problem) and the difficulty in manually evaluating models of embedded systems, we hypothesize that it is possible to reach a middle-ground solution: a partially-automated process of oracle generation which is capable of improving the testing activity of embedded systems by reducing its cost and increasing its effectiveness compared to manual evaluation, given their already described characteristics.

This research addresses the oracle problem in model-driven development, specifically for Simulink, a graphical system modeler and simulator widely adopted in industry. The oracle problem and its limitations are evaluated in this thesis to provide a basis to the proposition of an oracle engineering foundation which may conduct to the definition of partially automated oracle generators for Simulink-like models.

1.1 Motivation

Embedded systems have become increasingly sophisticated and their failure may be critical in many domains. Since it is usually impossible to guarantee error-free systems, the relevance in the search for gaps and improvements in the testing activity is evident.

Many partial solutions on the oracle problem have been proposed, but they are rarely treated as a whole. The tester should concentrate his/her efforts in planning the testing activity, not in researching ways to integrate isolated solutions and different technologies in a functional framework.

Simulink and free alternatives, as XCos, are a de-facto standard for the design and simulation of embedded systems in many different domains. It is widely known that, as later an error is discovered, more expensive is the cost to correct it. Therefore, it is preferable to eliminate defects previously in the design stage then in the later steps.

Also, Simulink-like executions may generate large amounts of output. Oftentimes, the testers assess the model correctness manually for a given test set, but this process is usually slow, error-prone, and very expensive.

The motivation is to contribute to the embedded software development process by defining an engineering process and developing supporting tools that allow the creation of automated oracles for Simulink models.

1.2 Objectives and Rationale

This research aims to provide an approach to encompass the needs and vulnerabilities previously described with a process, method, procedure and a tool, enablers of a partially automated oracle generator for Simulink models. Thus, the generation of test oracles for such models may be planned by the tester without the need to expend time and efforts to research technologies and existing solutions, as ways to integrate them.

In addition, this thesis contributes with concepts that lead to an oracle engineering, as well as the development and validation improvement of embedded systems supported by Simulink-like modelers, in terms of time, cost and quality of results.

In contrast to manual approach, automated oracles can produce more reliable and faster reports by allowing a greater amount of comparisons, covering more outputs, eliminating error-prone manual analysis, and detecting errors with more accuracy. Furthermore, testing cost can be reduced and productivity can be increased by comparing and reporting errors faster than if done manually.

Simulink models simulate dynamical systems which are defined as a set of possible states, together with a rule that determines the present state in terms of past states (Alligood et al., 2000). Specifications based on temporal logics are often used to describe allowable sequences of events (Baresi and Young, 2001). But the manual evaluation of a model based on such specifications may be infeasible due to the large amounts of output generated in a simulation. Hence, an automated comparison is even more opportune to dynamical systems in which temporal properties must be represented.

1.3 Contributions

As result of this thesis, the expected contributions are: (i) a process to define oracles for Simulink models; (ii) a method to generate specifications with high expressiveness (including temporal property representation) and interpretable by an automated oracle; (iii) a definition of an oracle procedure capable of analyzing a model execution with respect

to the given specification; (iv) a tool to provide empirical evidence of the viability of our proposal; (v) extensibility of our proposal to other tools similar to Simulink, as XCos and Scicos.

1.4 Thesis Outline

Chapter 2 provides an overview of the background concepts required for the understanding of this thesis. **Chapter 3** is based on a study (Nardi and Delamaro, 2011) which summarizes existing researches on the oracle problem, as well as possible gaps, approaches and limitations.

Chapter 4 defines a process to the generation of an oracle for Simulink-like models, together with a method to specify the information of interest and procedures to analyze results with respect to the specification.

Chapter 5 presents an overview of Apolom, an oracle generator tool implemented to assess the viability of this research, which supports all the process of the oracle generation. It is also presented a running example with Apolom to illustrate a complete definition of an oracle generator.

Chapter 6 details the implementation and limitations of Apolom to provide a perspective of the feasibility of the proposal and the obstacles to be overcome.

Chapter 7 brings four evaluations to assess the soundness of our proposal. It analyzes (i) whether the oracle specification is practical; (ii) whether the oracle affects the original model; (iii) whether the analysis time and cost are acceptable; and, (iv) whether the oracle is effective.

Chapter 8 presents a comparison between our proposal and available approaches and tools, to emphasize the innovative part of the research. Finally, **Chapter 9** concludes this thesis, presenting the conclusions and revising limitations and future works.

Chapter

2

General Concepts

Definitions of Software Engineering terms may vary in the literature and different research groups may adopt divergent meanings for such terms. This chapter presents the basic concepts to enable a common vocabulary for the understanding of this thesis.

Section 2.1 presents basic software testing concepts. Section 2.2 focuses on test oracles. Section 2.3 introduces Simulink. Section 2.4 concludes this chapter.

2.1 Software Testing

Testing is the process of executing a program with the intent of finding errors and aims to establish some degree of confidence that a program behaves as expected (Myers, 2004).

According to IEEE (2010), a **fault** is an incorrect step, process, or data definition in a computer program. An **error** is the difference between obtained and expected results. A **failure** is the termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. The distinction between error and failure lies in the visibility: a failure is a notable event. For example, an error can occur when computing the result of a function. If this error does not impact in the final result and the requirements are preserved, there is no failure.

In the testing process, a program is executed with input data and generated output data (obtained result) is compared with the specification (expected result). A test case is

a pair $(i, O(i))$, where i is an input data and $O(i)$ is the expected result for i. A test set represents all test cases used during the software testing.

This thesis focuses on the oracle problem. Therefore, data selection criteria and techniques will not be discussed in this chapter.

2.2 Test Oracles

Oracle is a mechanism which determines whether the output produced by a program is correct (Weyuker, 1982). This role is usually played by the tester.

An ideal oracle should be completely trusted (Mao et al., 2006a). The tester is error-prone and the effort to identify the correct result in a reasonable amount of time can affect negatively the test confidence which motivates the oracle automation research.

However, producing an automated oracle that maps all the possible input data to their respective expected results is as impractical as producing an error-free program. The following subsections describe the current approaches and categories that seek to solve the oracle problem with some degree of automation.

2.2.1 Oracle Approaches

An oracle automation approach can be roughly classified into two groups: pseudo-oracles and partial oracles.

A **pseudo-oracle** is a program written in parallel with the software under test, by a second team, following the same specification (Davis and Weyuker, 1981). Third-party components can be also considered, instead of double-coding (Hummel and Atkinson, 2005).

The pseudo-oracle output is considered as the expected result. Both are executed and their respective results are compared. The comparison can eventually include an acceptable margin of error. If there is a divergence between the results, the test fails.

There are a few limitations to the approach. Besides the burden of a double development, there is no guarantee that the oracle is correct. Therefore, if an error is detected, both must be debugged to verify in which one the fault is present.

Also, if the same fault is present in the oracle and in the software under test (SUT), it may produce the same wrong result in both executions and the comparison will not diverge, which will lead to a successful test, i.e., a false positive. Brown et al. (1992) suggest, in addition to different teams for the oracle and SUT developments, a different level of abstraction for the oracle implementation to reduce the chance that the same fault is present in both codes.

A **partial oracle** is capable of identifying whether the result is incorrect, even without the knowledge of the correct result (Weyuker, 1982). This is achieved by confronting the obtained results with constraints, as invariants, pre and post conditions (Kim-Park et al., 2009). In such case, the oracle identifies if the obtained result respects a set of rules instead of comparing its concrete value with some previously calculated expected result.

The approach presented in this thesis can be classified as a partial oracle, since it allows the tester to define constraints and relations between the data being processed in a Simulink model.

The terms passive and active oracles are also common in the literature. An **active oracle** mimics the behavior of a software component under test (Pasala et al., 2007). A **passive oracle** verifies the component behavior, but does not reproduce it (Shukla et al., 2005).

These concepts are transversal to the pseudo and partial oracles. A program written in parallel, i.e. a pseudo-oracle, reproduces the behavior of a SUT, therefore, it is also an active oracle. A pseudo-oracle that does not reproduce a result but compares it with a set of constraints is also a passive oracle.

Regardless the oracle approach, the primary function of an oracle is the verdict. This goal is fulfilled by analyzing some information and comparing it with the obtained result. The basic components of an oracle are the oracle information and oracle procedure.

An **oracle information** represents the expected behavior (Memon et al., 2003b), which is obtained from a specification, stored results, parallel program execution, metamorphic relations (discussed in the next section) or leaning machines. It can be a concrete behavior – the expected result – or an abstract behavior, as an acceptable boundary limit expressed by a constraint.

An **oracle procedure** compares the oracle information with the obtained result, which can be performed in execution time (on-line) or after it (off-line) (Durrieu et al., 2008).

Next subsections discuss these different representations of oracle information and how the oracle procedure uses it to report the test verdict.

2.2.2 Specification-Based Oracles

A formal specification offers an authoritative source of information about the correct behavior of a system (Baharom and Shukur, 2009) and it can be described in different levels of abstraction (Chen, 2002). Examples of specification languages are: Z notation, Object Z, VDM, JML, state machines, Petri nets, MITL and SDL.

A specification can be used as oracle information. In such case, the oracle procedure must be able to interpret the specification, which can require a compiler or interpreter, as well as a mapping between implementation and specification.

To avoid misinterpretation between concepts, three definitions are here discerned (based on IEEE (2010)): a **system requirement** is a condition or capability that must be met or possessed by a system, product, service, result, or component to satisfy a contract, standard, specification, or other formally imposed document. Requirements include the quantified and documented needs, wants, and expectations of the sponsor, customer, and other stakeholders; an **oracle requirement** is a requirement that must be evaluated by the oracle; a **system specification** is a structured collection of information that embodies the requirements of the system; and, **oracle specification** is a structured collection of information that embodies the oracle requirements.

Mapping: to compare the expected and obtained results, the oracle procedure must identify the relation between the implementation and the specification. There are many mapping approaches, as retrieve functions, embedded assertions and wrappers.

Aichernig (1999) describes an oracle scenario in terms of abstract data, concrete data and a mapping function (also called **retrieve function**): the implementation is executed with an input data, called concrete test input. A function r (Figure 2.1, adapted from Aichernig (1999)) maps the concrete input and output to its abstract representations in the specification, called respectively abstract input and output, by converting the implementation data (concrete) to the specification language data (abstract). A checker validates the abstract input and output according to predefined rules. The test passes if no rule is violated.

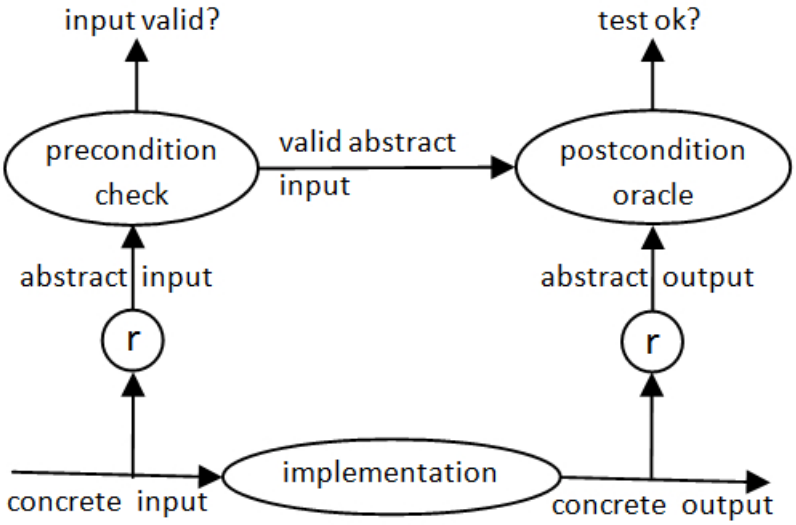


Figure 2.1: A test oracle scenario (Source: Aichernig (1999))

As an example of a test oracle process with retrieve function mapping, Jia (1993) proposed the following steps: (i) a compiler reads the specification and the retrieve functions, generating a test driver; (ii) SUT source-code and driver are compiled to generate an executable tester; (iii) test cases are applied to the executable tester and the oracle reports are produced.

A second mapping approach is related to the use of **embedded assertion** languages (Baresi and Young, 2001) which are expressions to be embedded directly in program code. They typically state properties to be checked at a particular control point in the program, therefore, acting as a partial oracle. There are programming languages, as Java, that natively support embedded assertions. Furthermore, there are specification notation languages, as Anna (Luckham and Von Henke, 1985) and JML (Cheon and Leavens, 2002)(Leavens et al., 2006), that allow the writing of requirements as assertions inserted within the code. In such cases, embedded assertions are mapped as reserved words, recognized by an interpreter/compiler, inserted in specific points in the code where they must be checked.

For example, a code recalculates a coordinate and has the following constraint: *the current coordinate can not be less than 5 before the calculation.* One may introduce an assertion in the SUT to ensure that an exception will rise if $coordinate < 5$, as presented at line 2 in the next code.

```
1 int coordinate = keyboard.nextInt();
1 assert coordinate>=5;
2 float new_coordinate = calculate_coord(coordinate);
```

A **wrapper** is a checker that surrounds the component under test (Edwards, 2001) without modifying its code. For example, given a class under test, a second one (the wrapper) is created with the same interface as the original but with other methods which are responsible to check some constraint. A test driver communicates with the wrapper which checks whether the class under test agrees with the specified (Figure 2.2, adapted from Edwards (2001)).

The representation layer is responsible by the data conversion from concrete to abstract and the abstract layer compares it with the constraints, as pre and post conditions.

The public methods of the original class are duplicated or overridden in the wrapper class. The wrapper controls the original class execution by calling its respective methods.

An example based on Shukla et al. (2005) is here presented: let us consider a class under test which represents a list of integers, with an *insert* method. A wrapper class inherits from the original class and overrides the *insert* method. Two other methods are

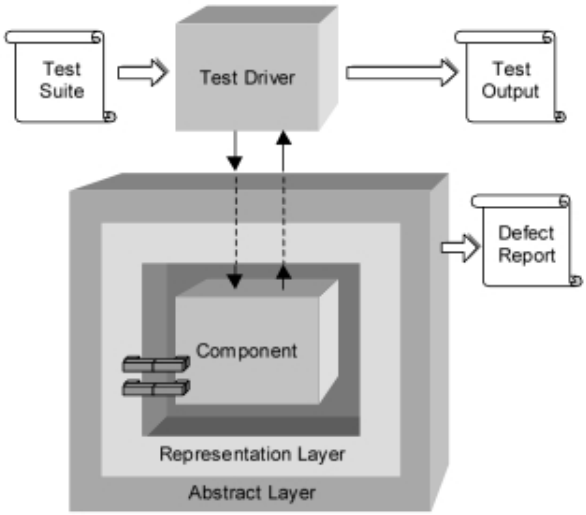


Figure 2.2: A wrapper (Source: Edwards (2001))

implemented in the wrapper: one identifies the number of elements of the integer list and the other compares the number of elements before and after the insertion. The method *insert* from the wrapper first checks the number of elements of the integer list by calling its respective method, then it calls the superclass *insert* method and, finally, checks again the number of elements of the list, comparing it with the first check. It is expected that the second checking is equals to the first check plus one. Otherwise, the element insertion has failed.

Specification paradigms: there are several specification languages and their respective paradigms reflect in many different oracle approaches. Examples of such specifications are briefly described next.

Executable models: if a model is executable, it can reproduce the SUT behavior. An oracle can use the model as oracle information and the simulation outcomes as expected results. An oracle procedure compares its results with the SUT outcomes and report an error if there is an unacceptable difference between them. For instance, Lasalle et al. (2011) apply a Simulink model as oracle information and its simulation calculates the expected results of a vehicle reaction with respect to the road characteristics when steering.

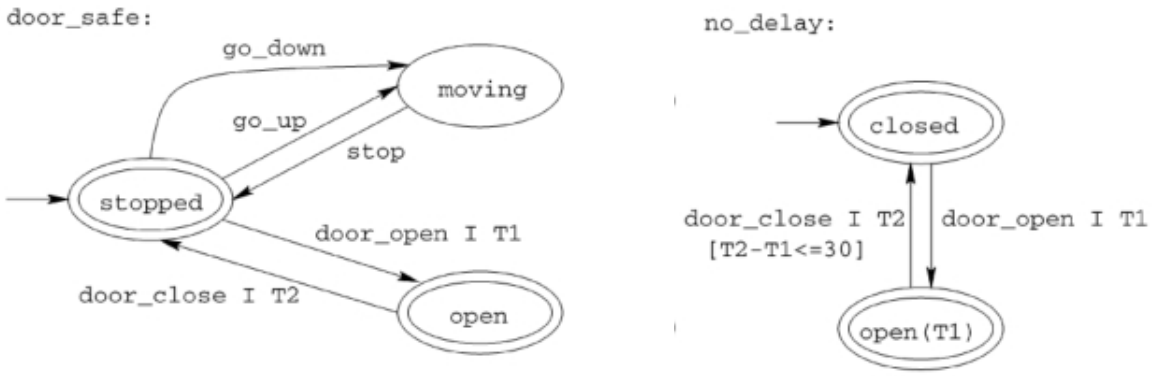
State machines: a parser can be applied to generate an analyzer from the machine; the SUT output is inserted into the analyzer; if the output is not expected in the machine, an error is detected. The state machine is the oracle information and the analyzer is the oracle procedure.

Andrews and Zhang (2003) give an example of state machines as oracle to test an elevator system. The output is stored into a log file. The requirements are: (i) the

door must be closed if the elevator is moving; (ii) the elevator must be stopped when the controller program terminates; (iii) the door must never be open for more than 30 seconds. The following log file is given:

```
call 3
go_up
reach 2
reach 3
stop
door_open 3 103325
door_close 3 103340
go_down
```

Two machines are created by the tester: one for requirements 1 and 2 (Figure 2.3a), and another for requirement 3 (Figure 2.3b).



(a) Requirements 1 and 2 **(b)** Requirements 3

Figure 2.3: State Machines (Source: Andrews and Zhang (2003))

An analyzer receives as input the state machines and the log file. If the analyzer can not recognize a line from the log or if there is no transition in the machine, the file is rejected and an error is identified. Eventually, the analyzer may be prepared to ignore valid log lines that are not represented in the state machines. In the example, *call* and *reach* are expected lines in the log file, but they are not represented in the state machines. In this case, they must be ignored by the analyzer instead of triggering an error.

Declarative paradigm: oracles in which the specification is written with a declarative language, that is, a nonprocedural language that permits the user to declare a set of facts and to express queries or problems that use these facts (IEEE, 2010). Embedded assertions, OCL (Briand and Labiche, 2001; Packevičius et al., 2007; Pilskalns et al., 2007) and Alloy (Svendsen et al., 2011) are examples of declarative languages.

Object oriented: a specification can be written following concepts of object-oriented paradigm, if the specification language allows it, as Object Z (McDonald et al., 1997; McDonald and Strooper, 1998; McDonald et al., 2003).

Temporal logic: such specifications allow reasoning about time. A temporal logic language may express time qualitatively, in which interval boundaries are defined in relation to other events, and/or quantitatively, so time distances can be measured.

Temporal logic languages are commonly associated with models as state machines or Petri nets. The former specifies temporal properties of a system and the models describe a system in terms of abstract model which simulates its behavior.

Wang et al. (2005) present a case study with requirements of a landing control system (LCS) of a lunar module, informally described as: "The occurrence of an error condition causes the system to switch to the emergency landing mode within 60 milliseconds". A safety property based on that requirement of LCS is written as: "When a TimeInt reaches Control System and the acceleration reading has not been completed, then Status changes to Emergency within 60 ms." The variable *TimeInt* represents a timer that interrupts the Control System if a predefined time is expired. *Status* represents the normal or emergency mode.

This safety property can be formally written in a temporal logic specification language, as *MITL*:

$$\Box_{[0,\infty]}((TimerInt \wedge \neg Done) \Rightarrow \Diamond_{[0,60]}(Status = Emergency))$$

The square symbol means "always in an interval" and the diamond denotes "sometime in the interval". The authors propose an algorithm that translates the MITL specification into a timed automata which is used as an oracle, as discussed on state machine oracles.

2.2.3 Metamorphic Relation Based Oracles

A metamorphic relation (MR) is an expected relation among the inputs and outputs of multiple executions of the target program (Chen et al., 2003). As example, an implementation of a sine function must always respect the following metamorphic relations: $sin(x) - sin(x + 2\pi) = 0$. In this example, the target program is the sine function and the difference between two executions (with inputs x and $x+2\pi$, respectively) is always 0. If a set of metamorphic relations is identified for a given program, then they can be used as oracle information. Such test is called metamorphic test.

The concept of metamorphic test is presented by Chen et al. (1998): "Although it is impossible to know if the output of the application is correct for arbitrary input, often these applications exhibit properties such that if the input or system state were modified in a certain way, it should be possible to predict the new output, given the original output".

With the previous example of sine function, even without the knowledge about the expected results, it is known that the presented relation must be true for any value of x. The tester defines test cases in which inputs are x and $x + 2\pi$. After executing the function with both

inputs, the tester subtracts the respective results. If the subtraction is different of 0 (or outside an acceptable boundary limit), an error is revealed.

2.2.4 Machine Learning Based Oracles

Machine learning are computational methods using collected data (or training data) to improve performance or to make predictions (Mohri et al., 2012). Supervised learning means that such collected data are previously labeled and predictions can be made for posterior unlabeled data.

Thus, given the prediction feature, supervised learning algorithms (SLA) can be applied as test oracles. In such approach, an SLA learns a behavior from test executions which are known as correct and becomes an active oracle. Three classes of SLA found in the literature are: neural network (NN), support vector machines (SVM) and info-fuzzy networks (IFN).

Neural networks are capable of simulating a software behavior (also called as *pattern*) from pairs of input/output (Shahamiri et al., 2009) and they are applicable as continuous (Jin et al., 2008) or discrete (Aggarwal et al., 2004) function approximators.

Two operations are involved with NNs: training and regression (or association, if used as classifiers) (Jin et al., 2008). Given a training set of input/output pairs, a NN is capable of finding the approximated function of a deterministic computational process. After trained, a network is capable of relating (classifying) outputs for inputs that were not used in the training operation.

The structure of a NN is composed by interconnected nodes (also known as *neurons*). The node structure is usually disposed in different layers, in which all nodes from a previous layer are connected to all nodes of the next layer. This organization allows a NN to approximate non-linear functions (Alpaydin, 2010).

NN oracles are justified in tests where the original program is unavailable (Vanmali et al., 2002).

SVM is used for classifying data objects and finding patterns in the classifications (Vapnik, 1995). It separates data from opposite classes with a hyper-plane located in the biggest possible margin with respect to the samples from each class which lies closest to it. As NNs, a SVM also depends on a training data, however, it consists of pairs of feature vector and its respective expected class. The SVM uses the training data to find separating hyper-planes to classify the feature vectors (Wang et al., 2011).

Info-fuzzy network is a directed rooted graph that represents a decision procedure (Agarwal et al., 2012). It involves data mining methodology, feature selection and classification. Feature selection is the process of identifying the components of the input vector that are most important in the decision process. IFNs have a single input layer, a variable number of hidden layers and a single target layer as a NN, but with a root node. Also, each IFN hidden layer represents an input attribute, not a weighted sum of input values (Agarwal, 2004).

2.2.5 N-version Based Oracles

An N-version based oracle comprises N independently written versions of a program, all conforming to the same functional specification (Eckhardt and Lee, 1985). If different outputs are produced by any version, a majority vote decides which output is probably correct.

M-mp (*m-model Program testing*) is a variation of N-version, where different versions of functions under test are implemented, instead of a complete system (Manolache and Kourie, 2001). The goal is reducing the test cost.

The paradigm and team diversity between versions and program under test aims to avoid correlated defects: the closer are versions and program, greater the risk of a same defect being present in more than one group (Manolache and Kourie, 2001).

Other similar approaches include: regression test, where older and stable versions, previously tested, are used as oracles to test programs developed by iterative processes (Memon et al., 2003b); and, the use of third-party components as test oracles (Hummel and Atkinson, 2005).

2.3 Simulink

This section presents basic concepts for the understanding of Simulink models. Most definitions were extracted from the official documentation [1].

Simulink is a block diagram environment for multidomain simulation and Model-Based Design which supports system-level design, automatic code generation and continuous test of embedded systems (Mathworks, 2013b). It is part of Matlab toolbox.

A block diagram is composed by blocks and lines. A **line** represents a mathematic relationship between signals. A **signal** is a time varying quantity that has values at all points in time. A **block** may be an action over a signal (Matsumoto, 2008), also called nonvirtual blocks, or an organizational element which plays no active role in the simulation (virtual blocks).

Blocks may be roughly classified as: **sources**, which provide input to other blocks; **sinks**, which receive output from other blocks; and, **operations**, which process signals. Table 2.1 presents a summary of common Simulink blocks.

Table 2.1: Simulink blocks

Block	**Library**	**Description**
Inport	Sources	creates input port for subsystem or external input
Constant	Sources	generates constant value
From File	Sources	reads data from MAT-file
From Workspace	Sources	reads data from Matlab workspace

[1] `http://www.mathworks.com/help/simulink/index.html`

Random Number	Sources	generates normally distributed random numbers
Outport	Sinks	creates output port for subsystem or external output
To File	Sinks	writes data to file
To Workspace	Sinks	writes data to Matlab workspace
Display	Sinks	show value of input
XY Graph	Sinks	display X-Y plot of signals
Product	Others	multiplies or divides its inputs
Gain	Others	multiplies input by a constant
Sum	Others	adds or subtracts its inputs
Logical Operator	Others	performs specified logical operation on inputs
Relational Operator	Others	performs specified relational operation on inputs
Switch	Others	switch output between first input and third input based on value of second input
Unit Delay	Others	delays signal on sample period
Integrate	Others	integrates signal

Figure 2.4 shows an example of Simulink model. It represents a braking detector (Chapoutot and Martel, 2009) with three subsystems. Figure 2.4a reproduces a breaking pedal as a mass-spring-damper system: given a system composed by a mass m attached to a spring with *spring constant* k and immerse on a viscous fluid of *damping coefficient* c, the oscillation in this system will be damped by the presence of the fluid. Such system may be represented by the following equation:

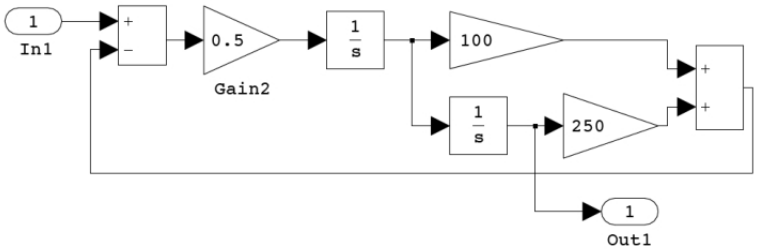
$$\ddot{x} = \frac{1}{m}(F - b\dot{x} - kx)$$

Where, $\ddot{x}$ is the acceleration, $\dot{x}$ is the velocity, and x is the position.

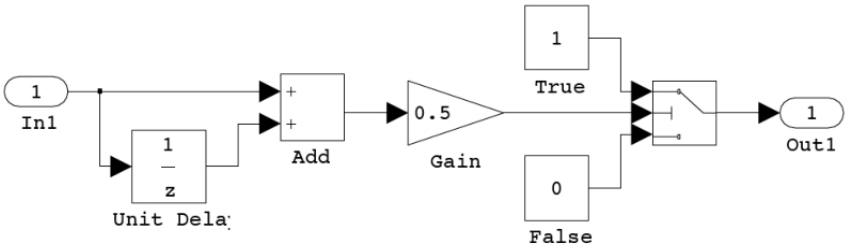
The model for such equation (Figure 2.4a) contains a *source* block (In1) which represents the force applied to the pedal and a *sink* block (Out1). Operation blocks perform the mass-spring-damper system: *Gain2* block represents $\frac{1}{m}$, where $m = 2$; second *gain* block represents the multiplication by the damping coefficient ($b = 100$); and, third *gain* block is the spring constant ($k = 250$).

Model from Figure 2.4b represents a detector of the force applied to the pedal. If the force is greater than a given threshold, breaking must be performed. And, Figure 2.4c reproduces the composition of the system based on the previous models. In this figure, Pedal and Control are virtual blocks. All output blocks within any subsystem are virtual because they do not play any active role in the simulation, as such as all inputs from the presented models.

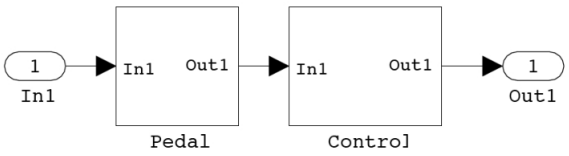
Simulink is part of Matlab and both environments may communicate. For example, it is possible to define a matrix on Matlab and configure *inport* blocks of Simulink to use such structure as data source. In the same way, it is possible to recover *outport* signals on Matlab.



(a) Breaking pedal



(b) Breaking control



(c) System

Figure 2.4: Breaking detector. Source: (Chapoutot and Martel, 2009)

Table 2.2 presents an example of simulation results from the introduced model. First column represents the simulation input data given by a designer, which represent the force applied to the pedal. Second column is the time steps and third column is the output, that is, whether the break action was detected.

Although this example presents a time vector with integer values, usually Simulink expresses time in intervals of 0.2 or 0.02 units.

The time of the next simulation step is determined by a component of the Simulink software called **solver**. There are many available solver algorithms and they may be classified as fixed or variable steps.

With **fixed-step** solvers, the step size remains constant while with **variable-step**, it may vary. If a designer plans to generate code from the model, he/she should choose fixed-step because one cannot map the variable-step size to a real-time clock. Variable-step solvers may be selected if it is intended to shorten simulation time, because the algorithm can dynamically adjust the step size as necessary, that is, if less time is needed to calculate the next step than a fixed step value, it will be shorten.

All blocks from Figure 2.4 are part of the default Simulink library. But it is possible to create new blocks and define their behaviors by the Matlab native language, or other languages as C/C++, Fortran, Ada (Matsumoto, 2008) and state machines. It is also possible to acquire

Table 2.2: Inport, time and outport

(a)	**(b)**	**(c)**
In1	***Time***	***Out1***
10	1	0
20	2	0
30	3	0
40	4	0
50	5	1
60	6	1
70	7	1
80	8	1
90	9	1
100	10	1
100	11	1
100	12	1
100	13	1
0	14	0
0	15	0

libraries (also called blocksets) from third-parties, as *UTRA FDD Blockset* from *Multiple Access Communications Ltd*, for 3GPP standard UMTS transmitter modeling; and *WLAN Toolbox* from *CommAccess Technologies*, for generating, demodulating, and decoding high speed WLAN waveforms compliant with IEEE Std. 802.11a/b/g.

2.4 Final Remarks

An ideal test oracle should always be able to assert the correctness of a SUT, given an input. However, it is not possible to develop automated oracles with such characteristic for all systems.

An oracle may be developed from several approaches with different information representations and a procedure that compares such information with the SUT results.

The application of these approaches varies according to the test purpose and available resources. Pseudo-oracles based on n-version, machine learning and of shelf components are usually applied to regression test, when previous outputs, third-party components or other versions of the same program are available.

But such oracles may not be always practical if there is no available version/program or if the development cost is too high. In these cases, a possible solution is the use of partial oracles based on specification or metamorphic relations which do not generate the expected results, but identify if the obtained results agree with the specified.

Although Simulink may be used as an oracle to test the implementation, two facts motivate a complete and deep test activity for such models: (i) Simulink allows automatic code-generation

from models. Therefore, if a model has defects, they will be propagated to the implementation; and, (ii) model complexity may be great enough to require a test activity itself.

Chapter

3

Evaluating the Application of Test Oracles

Chapter 2 discussed basic concepts and different oracle approaches. This chapter presents the result of a study which aimed to identify a solid research base, limitations on oracle appliance and the existence of supporting tools. The study followed the approach proposed by Biolchini et al. (2007). The following questions were addressed:

1. What are the main studies on test oracles?
2. What are the limitations observed in using these oracles?
3. Are there tools that support such oracles?

Section 3.1 presents an overview of the planning and execution of the study. Section 3.2 discusses the results w.r.t. the identified works. Section 3.3 presents oracle limitations. Section 3.4 shows a table with a list of tools which support the oracle automation. Section 3.5 presents a list of all selected works and their classifications based on quality criteria discussed in the next section. Section 3.7 concludes the chapter.

3.1 Overview of Study Selection

For this study, five indexed repositories were used: IEEE, ACM, Springer, Scirus and Scopus. Specialists were also considered in this study. The string applied in the electronic searches was:

("automated oracle" || "test oracle" || "testing oracle" || "automated oracles" || "test oracles" || "testing oracles")

The selection of the results was based on four criteria. Any paper which approaches at least one of them was included. The criteria are: (i) how an oracle may be generated automatically; (ii) how they may be defined, classified or applied; (iii) how oracles may be supported by tools; (iv) what are their limitations.

The selected works were also classified with respect to five quality criteria. Despite the name, the classification is not intended to generate a rank of relevancy in the Academy. The goal of such classification is providing a fast guide for posterior reference. The criteria are: (i) it mentions oracles for embedded systems; (ii) it mentions oracles for Simulink, Scicos or XCos; (iii) it presents comparison between different oracle categories; (iv) it mentions oracle categories; (v) it describes oracle limitations.

In the final selection, all pre-selected papers were fully read. Those which were not in line with any of the inclusion criteria were discarded. A total of 157 papers were selected.

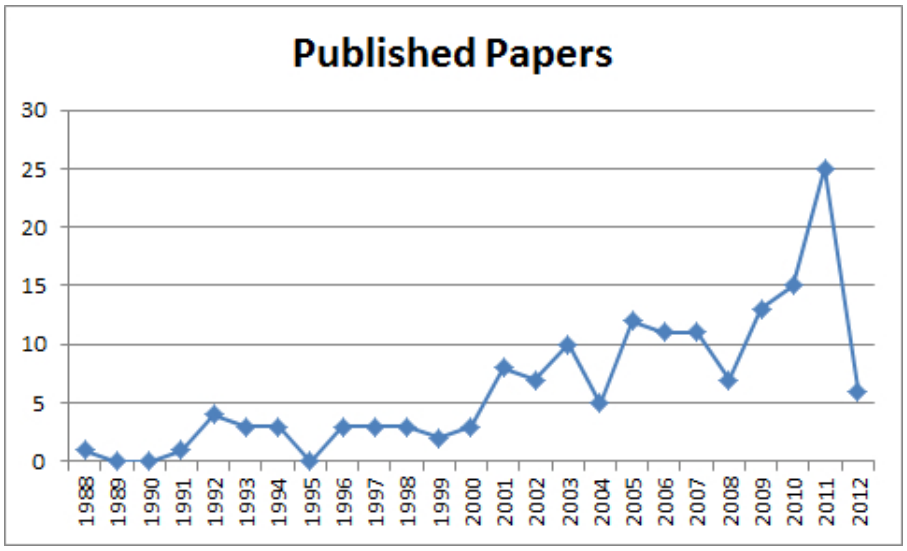


Figure 3.1: Publication by year

Figure 3.1 shows the relationship between years and publications. There is a heightened interest on research related to test oracles in the last 10 years, notably after 2001. In the last five years, 66 articles were published, representing 42.04% of the total published in 24 years.

Next subsections present the results by identified oracle categories, limitations and tools.

3.2 Identification of Oracle Categories

The identified oracle categories usually follow a taxonomy by oracle information, as presented in Section 2.2: specification-based, metamorphic relations, n-version and machine learning. This

Table 3.1: Relation between oracle categories

Category	Articles	%
Specification-based	109	69,43
Metamorphic relation	24	15,29
N-version or similar	14	8,92
Machine Learning	15	9,55

section presents statistical data extracted from the study and highlights the main distinctivity of each approach.

Table 3.1 shows the number of publications per category. There were articles that reported more than one class of oracles, therefore the category sum is not exactly 157 or 100%. Figure 3.2 presents the relationship between the number of articles and the year in which they were published, by oracle category.

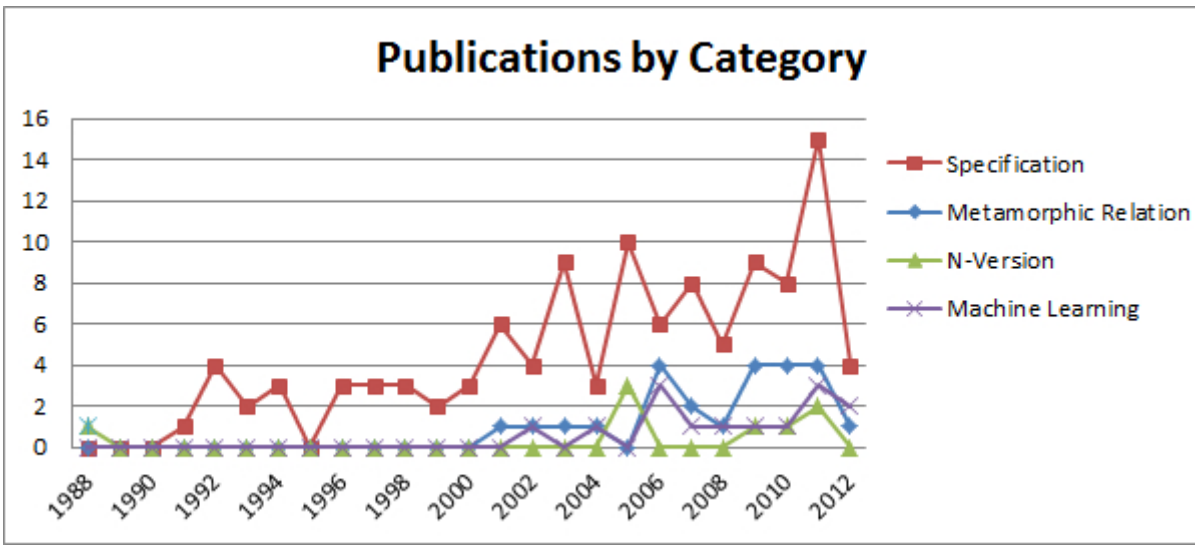


Figure 3.2: Publication by year, by category

Considering the inclusion criteria, and the sources used for the selection of articles, there are publications of specification-based oracles since 1991 with at least one publication per year except in 1995. The number of publications in the last five years is 41, which represents 37.61% of publications in the last 21 years.

Studies of oracles based on metamorphic relations began in 2001 with highest number of publications in 2006, 2009, 2010 and 2011. From the 24 published papers, eight have the same main author (Chen), from 2001 to 2003. This same researcher has co-authored works in 2009, 2010 and 2011 and four other papers have, as authors, researchers who participated as co-authors of Chen. The same network of researchers was responsible for 21 of the 24 published articles.

In relation to oracles based on machine learning, research production has increased, mainly after 2006. From the 14 papers, four deal with continuous functions, 4 using the algorithm of backpropagation and 2 using RBF (Radial Basis Function). One paper proposes the use of multi-networks oracle: a set of neural networks working in parallel as a multi-network oracle

may gain more than 1% of precision and 10% less misclassification error rate in comparison with a single-network oracle (Shahamiri et al., 2012). One paper describes the use of support vector machines for reactive systems. Three papers describe approximators of discrete functions with the algorithm of backpropagation and one with a SOM algorithm (Self-organizing map). Three articles, from 2006 to 2007, belong to the same group of researchers and discuss approximators of continuous functions with backpropagation and RBF.

Next subsections discuss each category results.

3.2.1 Specification-Based Oracles

This subsection summarizes the collected data from specification-based oracle researches, including a brief description of different languages and approaches found in the study.

Table 3.2 lists several approaches to specification-based oracles and their number of papers. The sum of related works does not match its total because there were works that referenced more than one specification.

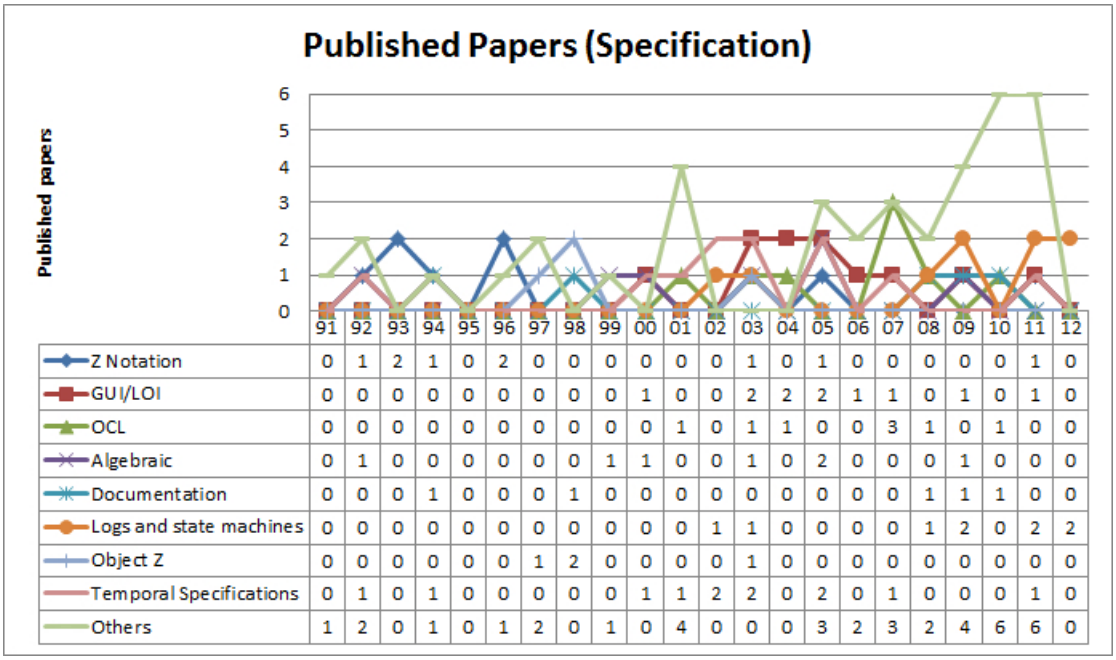
Table 3.2: Specification-based oracles

Temporal Specs.	12	GUI/LOI	11	Logs and statecharts	9
Z Notation	9	OCL	8	Algebraic Specs.	7
JML	6	Java assertions	5	Documentation	5
Model Transformation	3	Model Checking	2	VDM	2
Test Conditions	1	Anna	1	EASOF	1
Bit Wrappers	1	ASML	1	ConcurTaskTrees	1
CREOL	1	CTL	1	Dual Language	1
Complete Test Graphs	1	Eiffel	1	Prolog	1
Message Sequence Charts	1	IORL	1	Java Assertions	5
RSL/RAISE	1	SDL	1	TTCN-3	1
Prosper	1	WS-CDL	1	Esterel	1
TCL	1	ADEPT	1	SWRL	1
OSEK/PROMELA	1	Simulink	1	CTT	1
UML (sequence and interaction)	1				

GUI/LOI is not a specification, but a classification of categories about oracle information and oracle procedure in relation to levels of detail in the construction of oracles for testing GUI applications (Graphical User Interface). They were included in this study because such specification format can be easily adapted to represent any kind of constraints. The number of papers is the largest in quantity and distribution over the years. The same researcher participates as author in 90% of the published papers, being 20% as co-author.

Figure 3.3 shows that different specifications (grouped as Others) with no more than two publications have been proposed for use in oracles, since 1991, notably in the last years.

From specifications with more than two publications, Z Notation and state machines have been the most referenced both in absolute numbers and in distribution over the years. Three articles about Z (33,33%) were written by the same authors.



	91	92	93	94	95	96	97	98	99	00	01	02	03	04	05	06	07	08	09	10	11	12
Z Notation	0	1	2	1	0	2	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
GUI/LOI	0	0	0	0	0	0	0	0	0	1	0	0	2	2	2	1	1	0	1	0	1	0
OCL	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	3	1	0	1	0	0
Algebraic	0	1	0	0	0	0	0	0	1	1	0	0	1	0	2	0	0	0	1	0	0	0
Documentation	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0
Logs and state machines	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	2	0	2	2
Object Z	0	0	0	0	0	0	1	2	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Temporal Specifications	0	1	0	1	0	0	0	0	0	1	1	2	2	0	2	0	1	0	0	0	1	0
Others	1	2	0	1	0	1	2	0	1	0	4	0	0	0	3	2	3	2	4	6	6	0

Figure 3.3: Publications by year (Specification-based oracles)

The first two studies on the use of state machines as oracles belong to the same authors and represent 22,22% of the total, from 2002 to 2003. Research about such specification has increased in recent years, however, the focus usually is not the oracle. For example, the main goal of Yang et al. (2011) and Zhang et al. (2012) is addressing the problem of path infeasibility in the process of test case generation on EFSM models. During the model execution, the outputs associated with the collected test data can be used for the construction of an automated test oracle. In both works, not many details are given about the oracle procedure.

OCL (Object Constraint Language) is an extension of UML proposed by the Object Management Group (OMG), to allow the definition of constraints. OCL can be used, for example, to set limits of values to variables or pre and post-conditions of methods. Two research groups are responsible for 50.00% of the published papers. The other articles were published by different authors.

From the papers about algebraic specifications, 28.57% belong to the same group of researchers, with a publication in 1992 and one in 2000.

With respect to documentation-based oracles, 60.00% of the publications belong to the same research group from 1994 to 2010, 40.00% belong to a second group from 2008 to 2009, continuing the work of the first one.

All papers on Object Z belong to the same research group.

From the 12 papers that discuss specifications with support for temporal logic, the following languages were listed: **MITL** (*Metric Interval Temporal Logic*) (Wang et al., 2005), **TRIO** (Hakansson et al., 2003) (Lin and Ho, 2001), **EAGLE** (Goldberg et al., 2005), **Graphical Interval Logic** (Richardson, 1994), **RTIL** (*Real-Time Interval Logic*) (Richardson et al., 1992); Wang et al. (2003) use a modified version of Z Notation for specifying temporal logic; two articles

address the use of **Lustre**, (Bouchet et al., 2008) and (Durrieu et al., 2008); one article (Lin, 2007) references **ESML** (*Embedded Systems Modeling Language*) and **SIML** (*System Integration Modeling Language*); Lin and Ho (2000) use **time Petri nets**. In Barbosa et al. (2011), **CTT** (ConcurTaskTrees) is applied as specification to test GUIs against erroneous user behaviors. The high expressiveness of temporal specifications and the dynamical characteristic of Simulink models made such group of languages specially notorious to our research, which is discussed through this dissertation, mainly in Section 4.4, Section 4.2.1 and Chapter 7.

Table 3.3: Last years publications

	Total	Five Years	Two Years
Temporal Spec.	12	08,33%	08,33%
GUI/LOI	11	18,18%	09,09%
State Machines	9	77,78%	44,44%
Z Notation	9	11,11%	11,11%
OCL	8	25,00%	00,00%
Algebraic	7	14,29%	00,00%
Documentation	5	60,00%	00,00%
Object Z	4	00,00%	00,00%
Others	38	47,37%	15,79%

Table 3.3 represents the percentage of published papers in the last five and two years, respectively. There was one published work on Z notation in the last five years. There was also only one paper on algebraic specifications published in the last five years. And there were no published papers on Object Z in the last seven years. Results that have less than three publications which do not have relevant characteristics to embedded systems were grouped into "other specifications" category during the process of data compilation.

3.2.2 Metamorphic Relation Based Oracles

Chen et al. (2002) present the application of metamorphic testing based oracle on a case study to solve elliptic partial differential equation. The relationship identified can be used in other numerical methods. Chen et al. (2003) give the same concept applied on (Chen et al., 2001b), but with an example of a function for calculating power.

Gotlieb and Bernard (2006) applied the concept of exploitation of symmetries and random testing in a framework that contains a semi-empirical model. This model helps to decide when to stop testing and how to measure the quality of this test for JavaCard API, a technology that allows applets to run on SmartCards and other devices of limited memory.

Mayer and Guderlei (2006b) use seven metamorphic relations to test image processing operations by way of Euclidean distance transformation. The same authors (Mayer and Guderlei, 2006a) describe an empirical study on metamorphic testing with the use of Java applications that calculate the determinant of a matrix. In conclusion, the authors suggested four rules:

metamorphic relations that are in the form of equalities are especially weak; if the relation is an equation with linear combinations on each side and at least two terms to one side, then it is not vulnerable to erroneous additions but it is vulnerable to erroneous multiplications; typically good metamorphic relations contain much of the semantics of the software under test; metamorphic relations similar to the strategy used for implementation are limited.

Hu et al. (2006) conducted an experiment to investigate the cost effectiveness of using metamorphic testing. The authors use thirty-eight graduate students and three open source programs. As a result, they concluded that metamorphic testing is efficient and has the potential to detect more failures than the method with assertion checking.

Zhang et al. (2009), which are in the same research group of Hu et al. (2006), present the same experiment. The three programs are: *Boyer*, which returns the index of first occurrence of a pattern in a *string*; *BooleanExpression*, which validates boolean expressions; and, *TxnTableSorter*, an office application. Questions investigated in this paper, and the answers are: can students appropriately apply metamorphic testing after being trained? Yes. Can they identify correctly and usefully metamorphic relations to the target program? Yes. Can the same metamorphic relation be discovered by multiple students? Yes. What is the effort in terms of cost, in applying metamorphic testing? According to the results, metamorphic testing has the potential to detect more faults than *assertion checking*. On the other hand, may be less efficient in terms of cost. In general, students identified a greater number of assertions than metamorphic relations, although the number of metamorphic relations and assertions identified by students varied significantly. The authors believe that metamorphic testing helps developers to increase the level of abstraction better than assertions.

Ding et al. (2010) present the application of metamorphic testing on an image processing program used to reconstruct 3D structure of biology cells. As example of metamorphic relations, the tester adds mitochondria with different shapes to the cell images so that the 3D structures of these new mitochondria can be built. Then, the 3D structure of those new added mitochondria should be built as expected, the original 3D structures should not be changed, and the volume of mitochondria is expected to increase. The same author (Ding et al., 2010) applies metamorphic testing on a parallel Monte Carlo modeling program.

Sun et al. (2011) propose a framework of metamorphic testing for web services and present a case study with an electronic payment service as subject program to show its feasibility. It does not compare the results with other oracle classes, including manual comparison. The evaluation is set using mutation score.

In Xie et al. (2012, 2011), the authors apply metamorphic testing as oracle on spectrum-based fault localization strategy. A key concept applied to this strategy is *program spectrum*, which is a signature of some aspect that characterizes a behavior of a program. Program spectra are used on such strategy to identify the behavioral differences between old and new versions. For example, path spectra trace the set of single loop intraprocedural paths as program executes. Difference in program spectra between versions may indicate fault locations. The validation of oracles on spectrum-based faults was executed with subjects considered not large-sized programs.

Murphy et al. (2011) use metamorphic relation on oracles for health care software simulation.

3.2.3 Machine Learning Based Oracles

Vanmali et al. (2002) use an algorithm of backpropagation and state that such network can be trained by outputs from older stable versions of a program. A trained network can be used as an oracle to evaluate the correctness of new versions of a software, by simulating a model behavior, even though that model cannot guarantee 100% correction over the original program. It may be useful when older versions or trials are available for limited time or when their executions are too costly.

Aggarwal et al. (2004), Chan et al. (2006) and Jin et al. (2008), address the use of neural networks as oracles in problems involving classification. Two of these articles present as a case study, an oracle for triangle classification into isosceles, scalene, equilateral or not a triangle.

Mao et al. (2006b) and Mao et al. (2006a) apply neural networks to test statistical software. The authors assume that the relationship between inputs and outputs of an application under test is, in nature, a function. The appeal of using neural networks is the ability to approximate a function of any accuracy without the need to know the function.

Lu and Ye (2007) use RBF (Radial Basis Function) for construction of oracle similarly to Mao et al. (2006b). Sangwan et al. (2011) also apply RBFs with the objective of finding whether it can be used as a test oracle. They employ a triangle classification problem as experiment subject, as in (Aggarwal et al., 2004), (Chan et al., 2006) and (Jin et al., 2008).

Shahamiri et al. (2010) use a feed-forward with backpropagation algorithm to simulate logical software modules. The application used as case study was a registration-verifier. It is stated that different thresholds define the oracle precision and influences on the oracle accuracy. As higher the threshold, higher is the oracle precision. However, higher thresholds can make the oracles point a faulty output as correct. In this sense, as higher the threshold, higher the chance of faulty outputs being classified as expected outputs, therefore the oracle accuracy may decrease. Lower thresholds can make the oracle point a correct output as a faulty one. The same authors (Shahamiri et al., 2011, 2012) apply multi-network oracles, i.e., several standalone ANNs integrated to automate the mapping between input and output. The case study subjects were a web-based car insurance application and a student registration-verifier application (evaluated with a golden and mutation versions of both programs). In the comparison with a single ANN, they showed that the new approach is significantly more accurate. The experiment demonstrates that a multi-network oracle needs less effort to learn, which results in a easier training process w.r.t. a single network because the complexity of the SUT is distributed among the several ANNs.

Wang et al. (2011) apply support vector machine as supervised learning algorithm (SLA) to test reactive systems. In a first step, user guidance or assertions can be used to collect verdicts to test traces. Such traces are converted into feature vectors to train the SLA, which is used

as a test oracle. In their experiment, statements (not presented in the paper) are inserted into the SUTs to collect the verdicts. Also, bugs were implanted into the SUTs to check the correct verdicts of the test oracle. The results show that the proposed technique incurs little burden and overhead. The training time varies between 5 and 42 seconds and the testing time varies between 2 and 29 seconds. The correct verdicts fluctuate between 92.88% and 96.52%. As future work, they intend to clarify what basic features are needed for the testing of general reactive systems.

Agarwal et al. (2012) compare ANN (backpropagation/gradient descent) and info-fuzzy networks (IFN) as test oracles. They use ROC analysis, training time and dispersion analysis to conclude that IFN outperforms ANNs on faults and training time. They indicate IFN for testing initial, less stable versions of a given application. And no significant difference was found in classification accuracy and dispersion between ANN and IFN. As future work, they plan to include different network architectures to the comparison.

3.2.4 N-Version Based Oracles

Shimeall and Leveson (1988) use the N-Version concept on programs of combat simulation written in Pascal. Manolache and Kourie (2001) claim that M-mp, a variation of N-Version, provides low cost based on the justification that a program model do not need to be equivalent to the main program and only the function under test is implemented.

The idea of comparing results between two or more implementation can be extended to programs that already exist. A golden version of a program can be used as an oracle, for example in regression testing, component harvesting (Hummel and Atkinson, 2005) or "Multiple-implementation Testing" (*MiT*) (Taneja et al., 2010).

Tsai et al. (2005b) and Tsai et al. (2005a) propose a technique of majority voting to test a large number of Web Services (WS) that already exist and belong to a single specification to determine the oracle.

Hummel and Atkinson (2005) and Janjic et al. (2011) propose the creation of oracles from the same basic technologies that can be used to find components for reuse (such as *Extreme Harvesting*). Thus, it uses the components found in the searches combined as a pseudo-oracle to measure the confidence of built components.

3.3 Limitations

This section discusses limitations on the use of test oracles found in the literature. An issue shared with all kinds of oracles is the generalization problem. Even with large scale case studies, it is recurrent the difficulty in generalizing experiment results. Two ways are usually taken: proposals of future work to better evaluate the generalization (Agarwal et al., 2012; Sun et al.,

2011; Wang et al., 2011), or stating that the evaluation should investigate the potential of the technique rather than providing a statement of general effectiveness (Fraser and Zeller, 2012).

3.3.1 Limitations of Specification-Based Oracles

Nadeem and Jaffar-ur Rehman (2005) point that if the specification is incorrect, then demonstrating that the implementation conforms to the specification will not be of much use. Still, the functional specifications usually describe what the system needs to do when valid entries are given or certain conditions are met, but they usually omit a description of what the system should do when an invalid input is given, which results in the fact that only positive tests are performed. Similarly, for oracles based on algebraic specification, Bagge and Haveraaen (2009) indicate that the tests are as effective as the axioms on which they are based.

Machado et al. (2005) state that producing a specification with the correct level of abstraction is crucial in specification-based testing. But producing a correct, consistent and complete specification is difficult.

Bieman and Yin (1992) mention that an error in the specification will be propagated to the implementation in case the programmer uses the same specification to develop the implementation. As the oracle may be the tester or an automated tool (an implementation), in both cases errors may be present. Oracles can reduce performance when embedded in the code, but removing them after the test phase can cause unexpected problems such as changing in some reaction time, which may be critical for real-time systems.

According to Peters and Parnas (1994), there are restrictions on writing a documentation in order to be used as an oracle. As an example, the use of primitive relational operators like "=" is valid only for basic data types. For more complex types such as structures and objects, the specifier should define "=" through an auxiliary predicate, such as *absTypeEqual()*, to validate the equality to the abstract data type.

Peters and Parnas (1998) elicited difficulties encountered on implementing an oracle: (i) the documentation used to generate the oracle can be almost as complex as the program under testing and must be checked carefully. Supporting this difficulty, Kim-Park et al. (2010) cite a trade-off between specification precision and simplicity; (ii) an oracle procedure is a non-trivial program, also needing to be checked carefully; (iii) finally, not all program behaviors can be easily specified and checked using the proposed method.

According to Tu et al. (2009), it is not possible to use state machines to describe recursive concepts. Another issue is that the set of states is not finite in some situations. It can also incur in an state explosion, preventing the use of this kind of specification.

In the approach of Kanstren (2009), there are limitations as the need for a user to check the model correctitude to validate it as an oracle. The author cites the lack of empirical studies on the use of state machines.

Andrews and Zhang (2003) include a limitation on the use of ADTs, which is the possibility of the code being wrong. Also, the ADT specification used in the paper was small and simple. For more complex specifications it may be difficult to write efficient log file analyzers.

In Mottu et al. (2008), the high complexity of a transformed model makes difficult the use of oracles that check the validity of an entire model at once.

Stocks and Carrington (1993) advocate the use of formal specification with Z Notation, but they do not discuss the problems of producing an oracle procedure. According to Gotlieb and Bernard (2006) and Gargantini and Riccobene (2001), a limitation is the high cost relative to the time spent in developing a specification based on notations such as Z. Coppit and Haddox-Schatz (2005) show that the expressiveness of an assertion language can significantly affect the cost of implementing it. There are certain issues that are raised in the programming code that simply do not exist in the field of formal specifications and vice-versa. As example, given the implementation, it can be useful to check if a list of objects is null before invoking a method to add an element in the list. But in many formal specifications, there is no concept of null objects.

3.3.2 Limitations of Metamorphic Relation Based Oracles

According to Chen et al. (2003), metamorphic relations that cause higher "difference between executions" tend to be better. But this concept was not explicitly set. More research should be conducted to provide more explicit guidelines. Chan et al. (2006) state the choice of metamorphic relations was based on experience of the testers which may be a bias to the results.

Murphy et al. (2009b) describe that metamorphic testing can be a manually intensive technique for complex cases. The transformation of input data may be arduous for large data sets or practically impossible for entries that are not in a human readable format. Comparing the outputs can be error-prone for large data sets especially if small variations in the results do not mean error indication or when there is non-determinism in the results. The framework presented by the authors does not support metamorphic relations as:

$$ShortestPath(A, C) = ShortestPath(A, B) + ShortestPath(B, C)$$

.

3.3.3 Limitations of Machine Learning Based Oracles

Neural networks do not test event flow (Shahamiri et al., 2009). According to Jin et al. (2008), the input data may not be easily represented for use in neural networks, as characters and strings. Still, different elements in the input vector may have unequal contribution to the network. Deciding the structure of the network as the amount of layers and neurons may not be easy. The selection of training sets from test cases is another key problem that must be considered carefully: it should be previously evaluated, which implicates the use of other oracles.

Lu and Ye (2007) conclude that the use of RBF is feasible as an oracle, but do not perform comparison with other neural networks.

Shahamiri et al. (2010) discusses the relation between the chosen threshold value and the neural network accuracy. It seems that there is no final answer to define an ideal initial network settings and it may vary between applications or application domains.

3.3.4 Limitations of N-Version Based Oracles

This approach requires multiple implementations of the system functionality, it has high cost, it does not test the flow of events and it is not reliable (Shahamiri et al., 2009). Shimeall and Leveson (1988) mention that N-Version is not a substitute for functional tests.

3.4 Test Oracle Support Tools

Any means of automation associated with oracles was considered as a tool, including specification language translators, development environments that support oracles and frameworks. Table 3.4 presents the list of tools, their descriptions and the papers where they were found.

Table 3.4: Oracle Support

Name	References	Description
Adept	(Giannakopoulou et al., 2011b)	Enables the development of executable specification for HAI (human-automation interaction) domain.
BZTT	(Miller and Strooper, 2003)	Tool that uses the specification to generate states for testing. It uses constraint solvers to search for a sequence of operations that reach every state.
CaslTest	(Machado et al., 2005)	Test tool with support to Casl specification based oracles.
Corduroy	(Murphy et al., 2009b)	Framework that converts metamorphic properties in testing methods that can be runned using assertions checking at run-time JML (JML runtime assertion checking).
DART	(Memon et al., 2003a)	Regression testing framework for GUI applications associated with oracle. The oracle information can be obtained from the execution of previous tests or specification (legal sequence of events).
Dresden OCL Toolkit	(Cheon and Avila, 2010)	Interprets OCL constraints from a UML model and generates AspectJ code.

Name	References	Description
DSMDiff	(Lin, 2007)	Computes the difference between specific domain models, in model transformation.
Extreme Harvesting	(Hummel and Atkinson, 2005)	Tool to find and collect pre-fabricated components for reuse from the Internet.
FineFit/ Kodkod	(Faitelson and Tyszberowicz, 2011)	FineFit translates a specification into a relational model that serves as an oracle for testing object-oriented systems. Kodkod is a Java library that implements a bounded relational constraint solver
IFAD VDM-SL	(Aichernig, 1999)	Set of tools that allows the interpretation and code generation of pre and post-conditions. It allows verification through oracles based on post-conditions.
InTOL	(Wang et al., 2011)	Collects test traces. Used in a machine learning-based oracle
JML toolset	(Araujo et al., 2011b)	A compiler of the toolset translates specifications into runtime assertion checking code, producing Java classes with executable assertions
JPaX	(Xie and Memon, 2007)	Runtime monitoring tool.
JPF	(Giannakopoulou et al., 2011a)	A verification framework which checks state models for Java bytecode
Jtoc	(Qu et al., 2011)	Uses java annotations and Java inner class to construct contracts
KeYGenU	(Gladisch et al., 2010)	Chain-tool. KeY is a static checker that can automatically prove properties. GenUTest is a capture and replay tool
LETO/ Ocasime	(Durrieu et al., 2008)	Leto: Lustre-Based Test Oracle. Ocasime: offers facilities for regression testing. Lustre: specification language. Off-line tests. Test Schemes describe the test objective. Schemes are composed of parameters, variables, computer help and expected result of the test (temporal logics).
LUTESS	(Bouchet et al., 2008)	Test Environment (temporal logic).
MaC	(Xie and Memon, 2007)	Runtime monitoring tool.

Name	References	Description
MD-TEST	(Baharom and Shukur, 2009)	Tool that uses two types of documents: MIS, which specifies a module by its observable behavior and MIDD that provides information on internal data structure of a module.
NeuronDot-Net	(Shahamiri et al., 2010)	Engine which can be used to build different types of neural networks.
ORSTRA	(Xie, 2006b)	Support tool that checks results from regression test.
PATHS	(Memon et al., 2000)	GUI testing tool with support to AI and formal model test oracles.
PGMGEM	(Shukla et al., 2005)	Testing tool that stores names of exceptions and uses them to generate code exception handlers in a test driver. A wrapper is proposed as an oracle for this tool.
PLASMA	(Goldberg et al., 2005)	Route plan generation system, based on model. A real-time verification based oracle is proposed for this system.
Protégé/ SWRLTab/ Jess	(Bai et al., 2011)	Protégé is an editor to create ontology models. SWRLTab compiles rules. Jess engine interpretes the SWRL (Semantic Web Rule Language)
Protest	(Hoffman and Strooper, 1991)	Set of Prolog programs that support the development of test scripts and their applications to test modules implemented in C.
Simulink	(Lasalle et al., 2011)	Simulink models are used as oracles
TAGS	(Brown et al., 1992)	Compiles IORL specification in Ada.
TAOS	(Richardson, 1994)	Test tool with support to GIL specification based manual oracles.
μTest	(Fraser and Zeller, 2012)	generates test suites for object-oriented classes. It is an extention of Javalanche, which uses JUnit
TEAGER	(Seifert, 2008)	Test environment that allows execution of state machines specifications.
TOG	(Peters and Parnas, 1994) (Alawneh and Peters, 2010)	Test oracle generating tool, from a relational specification of the program and tabular expressions.
TOM	(Silva et al., 2008)	Generates oracle specification based on state machines.
TOTEM	(Briand and Labiche, 2001)	System test methodology based on UML in which the information is derived from OCL.

Name	References	Description
TROT	(Hagar and M., 1996)	Testing tools that support the creation of test oracles. They check the correctitude of the equation implementation, based on Anna formal specification language.
T-Vec	(Kuhn and Okum, 2006)	Development environment with associated specification and verification method for critical systems.
Uimdriver	(Li et al., 2011)	It has a verifier which checks if the collected execution traces are consistent with sequence diagrams and IODs on temporal ordering of message interaction
USE	(Pilskalns et al., 2007)	Validation tool that checks the states of objects generated from class diagrams in relation to the OCL.
Warlock	(McDonald et al., 2003)	Prototype tool that supports a method for generating test oracles for programs in C++ using the Object Z specification language.

The identified tools are distributed by information category, as follows: 30 tools support specification based oracles, in which 2 have temporal analysis capability (LUTESS and TAOS); 1 tool supports metamorphic relation oracles; 4 tools support machine learning oracles; 5 tools assist regression test; and, 1 tool supports extreme harvesting oracles.

Many tools are used to evaluate some specific research topic and require great efforts to be applied as a whole. For example, NeuronDotNet can be used to build neural networks, but the effort to map it to a program or the adaptation to some specific domain or program language is neglected.

Three tools are distinct by their close relation to Simulink: LETO, TAOS and T-VEC.

LETO is a test oracle for airbus critical systems which uses regression test and assertion checks with Lustre programming language. Lustre has limited temporal support. TAOS supports *regression test* (with an oracle procedure called Diff_Checker), *range checking* for verification of ranges of acceptable outputs and a prototype of *GIL checker* (Richardson, 1994).

Both LETO and TAOS are applied to embedded system tests, however they are used on programming languages and are not focused on model-driven development. T-VEC is actually used for Simulink model development and is discussed in Section 8.5.

3.5 Quality Criteria Application

In this section, a list of all selected articles and their respective relations with the quality criteria is presented.

Table 3.5: Quality criteria application

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Machine Learning	N-Version or similar	Limitations
Aggarwal et al. (2004)									√		√
Agarwal et al. (2012)						√			√		√
Aichernig (1999)							√				√
Aichernig et al. (2009)							√				√
Alawneh and Peters (2010)							√				
Almog and Heart (2010)						√					
Andrews et al. (2002)							√				
Andrews and Zhang (2003)							√				√
Antoy and Hamlet (1992)							√				√
Antoy and Hamlet (2000)							√				√
Araujo et al. (2011b)							√				
Bagge and Haveraaen (2009)							√				√
Baharom and Shukur (2008)							√				
Baharom and Shukur (2009)							√				√
Baharom and Shukur (2011)							√				√
Bai et al. (2011)							√				√
Barbosa et al. (2011)							√				
Bieman and Yin (1992)							√				
Bouchet et al. (2008)				√			√				
Briand and Labiche (2001)							√				
Briand and Labiche (2002)							√				
Briand et al. (2003)							√				√
Brown et al. (1992)							√				√
Chan et al. (2006)									√		
Chan et al. (2007b)								√			√
Chan et al. (2007a)		√						√			
Chen et al. (2001b)								√			√
Chen et al. (2002)								√			√
Chen (2002)							√				√
Chen et al. (2003)								√			√

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Machine Learning	N-Version or similar	Limitations
Chen (2003)								√			√
Chen and Aoki (2011)				√				√			
Cheon and Leavens (2002)							√				
Cheon (2007)							√				
Cheon and Avila (2010)							√				
Cho and Lee (2005)		√					√				√
Coppit and Haddox-Schatz (2005)							√				√
Dan and Aichernig (2005)							√				√
Ding et al. (2010)								√			√
Ding et al. (2011)									√		
Durrieu et al. (2008)		√	√	√			√				
D'Souza and Gopinathan (2006)							√				
Edwards (2001)							√				
El Ariss et al. (2010)										√	
Engels et al. (2007)							√				
Faitelson and Tyszberowicz (2011)						√	√				
Fraser and Zeller (2012)							√				
Gargantini and Riccobene (2001)							√				
Giannakopoulou et al. (2011b)							√				
Giannakopoulou et al. (2011a)							√				
Gibson et al. (2011)							√				
Gladisch et al. (2010)							√			√	√
Goldberg et al. (2005)		√	√	√			√				
Gotlieb and Bernard (2006)								√			
Grieskamp et al. (2001)							√				
Hagar and M. (1996)							√				√
Hakansson et al. (2003)				√			√				
Hierons (2012)							√				
Hoffman and Strooper (1991)							√				
Hu et al. (2006)								√			√
Hummel and Atkinson (2005)										√	√

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Machine Learning	N-Version or similar	Limitations
Janjic et al. (2011)										√	
Jia (1993)							√				√
Jin et al. (2008)									√		√
Jin et al. (2009)									√		√
Jonsson and Padilla (2001)							√				
Kanstren (2009)							√				
Kim-Park et al. (2010)							√				√
Kuhn and Okum (2006)							√				√
Kuo et al. (2010)								√			√
Lamancha et al. (2012)							√				
Lasalle et al. (2011)					√		√				
Li et al. (1997)							√				√
Li et al. (2011)							√				
Lin et al. (1997)							√				√
Lin and Ho (2000)				√			√				
Lin and Ho (2001)				√			√				√
Lin (2007)		√	√	√			√				
Lozano et al. (2010)							√				
Lu and Ye (2007)									√		
Luqi et al. (1994)							√				
Machado et al. (2005)							√				√
MacColl et al. (1998)							√				
Manolache and Kourie (2001)										√	
Mao et al. (2006b)									√		√
Mayer and Guderlei (2006b)								√			√
Mayer and Guderlei (2006a)								√			
McDonald et al. (1997)							√				√
McDonald and Strooper (1998)							√				√
McDonald et al. (2003)							√				√
Memon et al. (2000)							√				
Memon et al. (2003b)							√				√

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Machine Learning	N-Version or similar	Limitations
Memon et al. (2003a)							√				
Memon and Xie (2004b)							√				
Memon and Xie (2004a)										√	√
Memon and Xie (2005)							√				√
Memon et al. (2005)							√				√
Meyer et al. (2007)							√				
Miller and Strooper (2003)							√				
Mottu et al. (2008)							√				√
Murphy (2008)								√			√
Murphy et al. (2009a)								√			√
Murphy et al. (2009b)								√			√
Murphy et al. (2011)								√			
Nadeem and Jaffar-ur Rehman (2005)							√				
O'Malley (1996)							√				√
Packevičius et al. (2007)							√				
Peters and Parnas (1994)							√				√
Peters and Parnas (1998)							√				√
Peters and Parnas (2002)							√				
Pilskalns (2004)							√				
Pilskalns et al. (2007)							√				√
Qu et al. (2011)							√				
Rajan et al. (2010)							√				√
Richardson et al. (1992)				√			√				
Richardson (1994)				√			√				√
Seifert (2008)		√					√				
Sangwan et al. (2011)									√		
Shahamiri et al. (2009)						√			√	√	√
Shahamiri et al. (2010)									√		√
Shahamiri et al. (2011)						√			√		√
Shahamiri et al. (2012)						√			√		√
Shimeall and Leveson (1988)										√	√

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Machine Learning	N-Version or similar	Limitations
Shrestha and Rutherford (2011)							√				
Shukla et al. (2005)							√				√
Silva et al. (2008)							√				√
Skroch (2007)							√				
Stocks and Carrington (1993)							√				√
Stocks and Carrington (1996)							√				
Sun et al. (2011)								√			
Svendsen et al. (2011)	√	√	√	√			√				√
Taneja et al. (2010)										√	
Tiwari et al. (2011)										√	√
Tsai et al. (2005b)										√	√
Tsai et al. (2005a)										√	√
Tu et al. (2009)							√				√
Vanmali et al. (2002)									√		
Wang et al. (2003)				√			√				
Wang et al. (2005)			√	√			√				
Wang et al. (2011)									√		
Xie (2006a)							√				√
Xie (2006b)										√	√
Xie and Memon (2007)							√				√
Xie et al. (2009)								√			
Xie et al. (2010)								√			
Xie et al. (2011)								√			√
Xie et al. (2012)								√			√
Xing and Jiang (2009)							√				
Yan (1999)							√				
Yang et al. (2011)							√				
Ye et al. (2006)									√		
Yoo (2010)								√			√
Zhang et al. (2009)								√			√
Zhang et al. (2012)							√				

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Machine Learning	N-Version or similar	Limitations
Zheng et al. (2011)										√	
Zhou et al. (2010)							√				
Zhu (2003)							√				

It is possible to note the lack of comparative studies between oracles. From 157 papers, only 6 have significant comparison or mention some taxonomy of test oracles where 3 are written by the same author. Also, 76 papers discusses their proposal limitations, which indicates that 51.59% of the presented works describe solutions but do not address their applicability deeply enough to provide an insight into their threats to validity.

Also, 10.8% of the researched works focus on embedded systems, where only 1.2% provides some limited temporal property validation, although they are not target on model-driven development.

3.6 Threats to Validity

When executing a literature review, the search string may not represent all the universe of papers related to the subject. In this study, the string restricts the number of papers to a manageable level ensuring that they approach the addressed questions. In this way, for example, even if state machines may be used to describe the oracle information, only papers which explicitly reference them as oracles were considered. Therefore, related works which do not point specifically to the oracle topic were excluded.

3.7 Final Remarks

This chapter presented a study to identify works based on three items of interest: types of test oracles, their limitations and support tools.

Any oracle that could be adapted to Simulink-like models was considered, totaling 157 selected papers. Four types of oracles were identified: specification-based oracles (109 papers), metamorphic relation oracles (24 papers), N-Version oracles similar approaches (14 papers)

and machine learning oracles (15 papers). From specification-based oracles, 5.73% refer to Z Notation, the same proportion to state machines, 7.64% to temporal logic specifications, 5.09% OCL, 4.46% refer to algebraic specifications, and 29.9% to other specifications with less than six publications.

From the recurrent limitations in any specification languages, there is an emphasis on the fact that if a specification is incorrect, this error will be propagated in the next stages of development. Other limitations include the level of abstraction: differences in the layers of abstraction between implementation and specification (or model) may require efforts to map between them regarding the lack of representation of concepts present in one layer but not in the other. It may be also difficult to express detailed specifications considering the time and cost for that. The higher is the specification expressiveness, closer to the implementation is the amount of resources spent to implement and verify it.

Limitations of oracles based on metamorphic relations include lack of guidelines for finding the relations and their choices are based on the experience of the testers. Their use can be also laborious for large systems. Regarding oracles based on machine learning, these are not applicable to streams of events or non-deterministic systems, and there is a lack of studies that indicate what type of network is best applied to different domains. Oracles based on N-Version are expensive given the need to create several versions of the system.

This chapter presented a list of 42 test oracle support tools in which none addresses large amounts of output neither enables expressive representation of temporal properties. Such tools usually do not provide support to all the oracle generation process but aids to evaluate some specific research topic. These findings evidence a gap which also motivates the development of a special-purposed solution which tackles the already described characteristics in this research.

Chapter

4

Designing an Oracle Generator

As earlier stated in Section 2.2, when the tester usually plays the role of oracle such manual activity is error-prone. Additionally, the effort to identify the correct result of a large set of outputs in a reasonable amount of time may affect the test confidence.

Although the production of a perfect automated oracle is as impractical as producing an error-free program, Chapter 3 showed that researches on oracle automation has increased in the last decades, with notable growth in the last years. Such researches point to a middle-ground in which several manual efforts may be replaced by automated means.

This chapter presents part of the core contribution of this thesis. It describes an oracle engineering foundation to the partially-automated generation of test oracles for Simulink-like models. The initial sources of information to the design of our proposal are described in Chapter 3. Later sources were also used to assess our solution novelty and soundness, which are described in Chapter 8.

Next sections present the oracle information structure, a mapping approach and procedures to the oracle analysis. Thus, this chapter does not address the use of specification language, which is discussed in the next chapter.

Section 4.1 presents an overview of our approach, with a process model. Section 4.2 describes how the oracle information is structured. Section 4.3 discusses the mapping between oracle information and model. Section 4.4 presents ways to analyze the simulation data with respect to the oracle information. Section 4.5 concludes the chapter.

4.1 Oracle Process

The oracle definition process behind our solution, presented in Figure 4.1, is composed by three steps: **information definition**, **mapping**, and **analysis definition**.

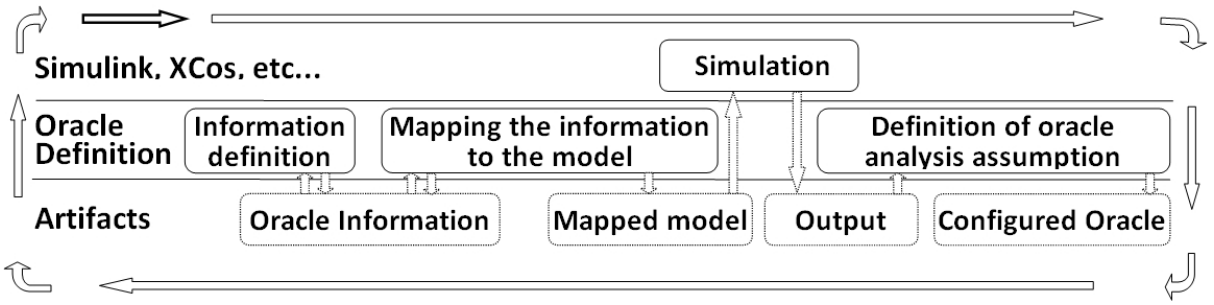


Figure 4.1: Oracle definition process

As previously discussed, the *oracle information* states what the expected outcome is and how it should be analyzed with respect to the input data. Because it is a partial oracle the information does not reflect the exact expected output but, instead, it determines constrains that the output must obey.

Definition 1 - *Information definition* *is the activity that provides the means to represent the oracle information.*

The oracle must be able to represent and interpret the model simulation outputs and inputs so that the results may be compared with their equivalent abstractions within the oracle information. Such oracle references (here called as *attributes*) at some point need to be mapped into the model. For instance, if the tester wants to evaluate that an *alarm* is on when *smoke* is detected, then: (i) both Simulink model representations must be identified – which signals represent *alarm* and *smoke detection* in the model; and, (ii) the oracle must be able to establish that, at each simulation instant, the values from such signals must be associated to their respective representations (attributes) described in the oracle information.

Definition 2 - *Mapping* *is the activity that enables the oracle to reference the model simulation results.*

An instrumented version of the model results from mapping attributes of the oracle information onto the model. The oracle analysis states how to evaluate simulation data against the information and how to store its results without inconsistencies entailed by possible future changes in the information. For example, after the oracle analyzes a simulation and generates a report, the tester may decide to change the evaluated requirement to test its results against some different constraint limit. In this situation, if it is important to keep a record from older

analysis, the report must store all relevant analysis data, including the analyzed data, the analyzed requirements and the verdict in the moment when the analysis was performed. Otherwise, when the tester decides to review an older result based on an updated requirement, it may reflect an inconsistent report because the older result was analyzed with a different requirement.

Definition 3 - *Analysis definition* *is the activity that provides means to allow the oracle to interpret the oracle information and analyze the model simulation results with respect to such information.*

It is worthy to note that such process is paradigm-independent. The oracle information may be represented by a specification language, a component with equivalent functionality, a neural network or metamorphic relations (information origins are discussed in Chapter 2). In any case, the information must be prepared and related to the model.

As example of a complete oracle definition process, the tester may specify the oracle information through a specification language and map its requirements into the model. The mapping generates an instrumented model which is executed within a simulation tool (as Simulink). The instrumented model dumps the mapped points of interest as sequences of values that are then retrieved by the oracle. The tester defines how the data must be analyzed and the oracle is ready to compare the simulation result with the specified requirements.

The process foresees incremental information definition and mapping, in such a way that the tester may define priority test requirements, map them, execute a test and, later, specify new sets of requirements to be tested. New requirements do not imply in new mapping: this step only needs to be executed on references to unmapped model inputs and outputs. After a cycle, all steps may be modified in a new iteration without losing consistency with previous result. For example, if a requirement is modified in a new cycle, the previous results may seem inconsistent because they were generated with a different requirement. This issue must be handled in the process (as it is described in Chapter 5).

Finally, the development of the early steps of an oracle generation should not impose the presence of a Simulink model, so the oracle information may be planned during or before the model development. Four scenarios were considered: (i) the tester does not have a high level specification (in this case, a Simulink-like model is the higher level specification); (ii) the tester has a high level specification, for instance, documents in natural language and a model; (iii) the tester has a high level specification but not the model; or, (iv) the project is starting and no model or requirement was yet formalized.

In the presented scenarios, two distinct combinations must be noted: presence or absence of a high level specification, and presence or absence of a model.

The absence of an specification means that the oracle information may be planned as the system specification or part of it. However, in the presence of a higher level specification, the oracle information must reflect part or all the system specification, which may require a careful verification and tracing between the oracle information and the system specification.

The absence of a model, as it was stated, should not restrict the first steps of an oracle generation. Moreover, the oracle generation could be used as a framework to a test-driven development, in which a model would be developed based on previously written test cases.

Next sections detail each step of the process.

4.2 Information Definition

An oracle information encompasses **attributes**, which provide an *abstract view* on the values of interest, and *rules*, which use the attributes to state the required behaviors of a system.

Definition 4 - ***Rule*** *is a representation of a system requirement which can be interpreted by the oracle.*

A rule can be any expression that relates to the data entering or exiting the model. Details are given in Section 4.2.2 and the definition of system requirement is available in Section 2.2.2, page 8.

Definition 5 - ***Oracle information attribute*** *is the representation of (i) a signal value, which is mappable onto a model's signal; (ii) an instant value; or, (iii) a simulation time, at a given oracle analysis instant.*

Definition 6 - ***Oracle analysis instant*** *is the current instant of the oracle analysis.*

Since signals may vary over time (Mathworks, 2012), a sequence of values is generated for each signal during a simulation.

Definition 7 - ***Sequence*** *is a numerical series that represents all the values of an attribute from the first to its last instant.*

To illustrate such definitions, it is used an example of a reactor's model with a signal that renders a boiler pressure, simulated for 500 instants. Assuming each instant represents 0.2 seconds, the simulation would reproduce 100 seconds of simulation. At least three sequences can be abstracted to the oracle information: the boiler temperature values during the simulation (each value is represented by Definition 5(i) and all values in sequence portray Definition 7); the instants from 0 to 500 (Definition 5(ii)); and, their respective simulation times from 0 to 100 seconds (Definition 5(iii)).

The contribution of the proposed solution for specifying oracles is twofold: it aims to provide suitable means to reason on events and on their relations, but it also wants to suggest a method to structure and organize an oracle definition, as it will be discussed in Section 4.2.2. Next section presents more details about how rules may be described, as well as their temporal properties.

4.2.1 Temporal Logic

Simulink models simulate dynamical systems. A dynamical system is a set of possible states, together with a rule that determines the present state in terms of past states (Alligood et al., 2000). Such systems evolve in time (Jost, 2005).

Specifications based on temporal logics are often used to describe allowable sequences of events (Baresi and Young, 2001). Temporal languages may express time qualitatively, in which interval boundaries are defined in relation to other events, and quantitatively, so time distances can be measured.

Qualitative operators provide ways to express properties like precedence, eventuality and invariance. Quantitative operators grant the ability to state that a given property will hold k time units from the current time instant (Felder and Morzenti, 1992).

TRIO (Ghezzi et al., 1990) is a first-order temporal specification language that provides qualitative and quantitative operators, as well as propositional operators and existential and universal quantifiers. A major goal of TRIO is the executability of specifications without giving up the expressiveness derived from temporal quantification (Felder and Morzenti, 1992). These characteristics led us to adopt this language to describe rules.

Any TRIO operator is derived from the basic temporal operator $Dist(A, t)$, which states that A must hold at instant t, where t may be negative, 0 or positive (for past, present and future instants, respectively). For example, the following formula defines $Futr$ operator:

$$Futr(A, t) \equiv Dist(A, t) \wedge t \geq 0$$

In this way, TRIO is not limited to a preset list of temporal operators. Instead, a great number of new operators can be derived from $Dist$, as:

$$UntilW(A_1, t_1, A_2) \equiv \exists t_2 (Futr(A_2, t_2) \wedge \forall t' (t_1 < t' < t_2 \implies Futr(A_1, t')))$$

The formula intuitively means that A_2 must hold in the future and A_1 will hold from instant t_1 until then. A partial and incomplete list of TRIO temporal operators is listed in Table 4.1. Given those operators, the following examples illustrate two requirements and their respective TRIO representations:

Example 1. A trivial system that given an input ($in(a)$) produces an output ($out(b)$) exactly 300 instants in the future can be specified as:

$$in(a) \implies Dist(out(b), 300)$$

To simplify TRIO implementation, a subset of this language was actually considered. For example, recursivity was not implemented.

Table 4.1: TRIO Operators

Operator	Description
$Always(A)$	A must hold in every time instant.
$Past(A,t)$	A must hold at intant t in the past, where t must be greater or equals to 0.
$SomF(A)$	A must hold sometime in the future.
$SomP(A)$	A must hold sometime in the past.
$Lasts(A,t)$	A must hold for the next t instants in the future.
$Until(A_1,A_2)$	A_2 must hold in the future and A_1 will be true until then.

New functions were also added to the language, as presented in Table 4.2.

Table 4.2: Added functions

Function	Description
$NowOn(A)$	returns the number of instants that A holds consecutively starting from the oracle analysis instant.
$NowOnTimes(A,t)$	returns the instant in which A holds for the t-th time, starting from the oracle analysis instant.
$Instant()$	returns the oracle analysis instant.
$Instants()$	returns the number of instants of the simulation under test.
$Clock()$	returns the simulation time at the oracle analysis instant.

The difference between $Instant()$ and $Clock()$ relies in the meaning of oracle analysis instant and simulation time. An oracle analyzes a rule at each instant (oracle analysis instant), starting from instant 0 until the last instant n ($n \in \mathbb{N}$). In this way, a simulation contains $n+1$ instants. Each instant represents a simulation time. For example, if each interval between instants is configured as 0.2s in Simulink, then instant 0 represents 0.0s, instant 1 represents 0.2s and instant 2 represents 0.4s of simulation.

Example 2. A more complex requirement with a function from Table 4.2 is given next: if the temperature of a system is greater than x for an interval of time (*interval*) of more than t_1 instants, then a red indicator must be on to indicate a critical situation until it lowers to a safer temperature. A safety protocol (*safety*) must be started within the next t_2 instants and should last the same number of instants as *interval*. Also, a yellow indicator must be on after the safe temperature is reached and until the alarm is turned off. These constraints can be specified as:

$$Starts(greaterthan(temp; x)) \wedge interval = NowOn(greaterthan(temp; x)) > t_1 \implies$$
$$Lasts(on(red), interval - 1) \wedge \exists(Lasts(on(safety), interval - 1), 0, t_2) \wedge$$
$$UntilW(on(yellow), instant() + interval, off(alarm))$$

The left part of connector $\wedge$, in the first line, identifies the first instant (the starting point) of an interval of occurrences in which temperature *temp* is greater than x. The right side of the connector has an operator to count how many instants the temperature is above x from the starting point and verifies whether it endures longer than it is allowed (t_1). If both conditions hold, then the second line guarantees that the red indicator is on for the same interval of time as the temperature is critical. It also verifies whether safety protocol is activated for the next t_2 instants in the future and that it endures for the same interval of time as *interval*. Line three verifies whether yellow indicator is turned on from the next instant where the temperature is below the critical point until the alarm is switched off.

Expressions may become complex to write. Methods to simplify and reduce temporal expressions can be applied, as presented in (Baresi et al., 2009). In addition, a "library" of expressions that represent standard behaviors in given domain might be very useful. Some of these behaviors have been represented as built in expressions in our tool (Chapter 5), as *greaterthan*, *lessthan*, *ascendant*, *descendant* and *geocoord*.

Next section discusses how rules may be structured.

4.2.2 Information Structure

Due to the complexity that a model may achieve, as well as its respective specification, it was decided to organize the oracle information around three layers: requirements, behavior and modularization.

The requirements layer is the core of the information structure and comprises the rules that state the requirements on a system. Rules are grouped into oracle information units (OIUs).

Definition 8 - ***OIU*** *is a set composed by a main rule and, possibly, constraints and safeguards.*

Definition 9 - ***Constraints*** *express rules that must be true when the main rule holds.*

Definition 10 - ***Safeguards*** *express what must be true if the main rule or constraints are violated.*

This organization is intended to alleviate the complexity of defining complex rules by providing a built-in organization with trigger-dependent relations between simpler rules. It also aims to alleviate the issue raised by Nadeem and Jaffar-ur Rehman (2005): considering the works on automated oracles, specifications usually are not used to describe what a system must do when invalid inputs are given. With an OIU, the tester may describe safeguard rules that must be checked if a main rule (or constraint) is disrespected, that is, if invalid inputs are given.

Definition 11 - ***Requirement layer*** *is an abstraction which provides a defined structure to design requirements as rules and a violation report criterion.*

The requirement of Example 2 is used as a base to the next explanation. In this example, an OIU can be written as a main rule with no constraints and safeguards as presented before, or it can be reorganized into smaller rules, if convenient.

For example, a main rule may state that the temperature should be always below a critical level. A safeguard (S_1) may state that, if the main rule is not respected, the *temperature* may not be above the critical level for more than a given interval. Another safeguard (S_2) may imply that, if the critical level holds for longer than the given interval, a safety protocol must be started within an acceptable delay (t_2) and lasts for a given interval. A constraint (C_1) may state that if the alarm is off, the green indicator must be on.

The organization into smaller rules allows the oracle to report either a **success** or different **types of violations**: (i) critical failure, in which the main rule and at least one safeguard are violated, (ii) constraint failure, in which the main rule holds true, at least one constraint does not and all safeguards are respected; (iii) constraint critical failure, in which the main rule holds true, at least one constraint does not hold and at least one safeguard is disrespected; and, (iv) non-critical failure, when the main rule is violated but all safeguards are respected.

The behavior layer provides recurring, readily-available, and reusable behaviors (as macros or components) for OIUs. As instance, one may be interested in knowing when values are greater than a threshold, or ascendant, or if they are functions of other values provided by other signals. These general-purpose rules help to factorize common problems and avoid re-stating them several times. They also contribute to the maintenance of the information since a rule used many times only needs to be changed once.

A predicate is a boolean valued function $P : X \rightarrow \{0, 1\}$ (Haji-Valizadeh and Loparo, 1994). In this thesis, behavior is defined as follows:

Definition 12 - ***Behavior*** *is a predicate which can be related to at least one sequence.*

Definition 13 - ***Behavior layer*** *is an abstraction which addresses the definition of specific behaviors that must be identified in a sequence during the oracle analysis.*

Definition 14 - ***Behavior-sequence pair*** *is a tuple* $\langle behavior, (seq_1, \ldots, seq_n)\rangle$*, where a behavior references one or more sequences. It establishes that the oracle must identify in which instants a behavior is true for a given sequence or sequences.*

Figure 4.2 presents a Simulink model of a boiler control in which the requirement of **Example 1** must be applied. In such case, the temperature should be always below a critical level.

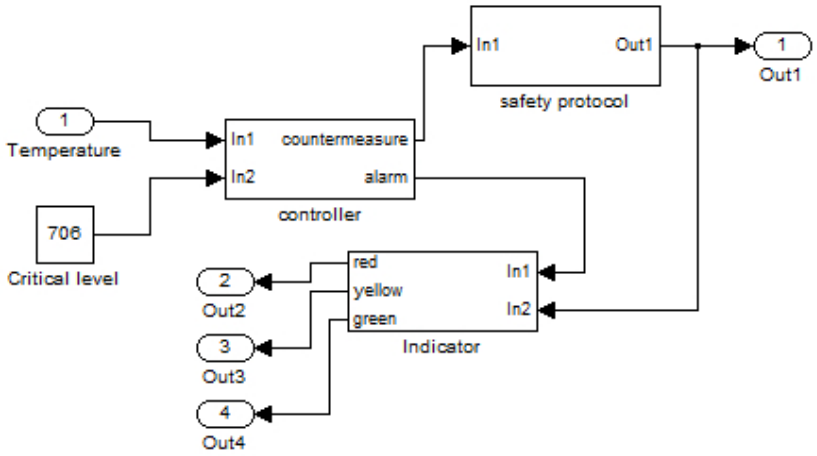


Figure 4.2: A model of a boiler control

Applying the described concepts, a behavior (*less than*) may be analyzed over two signals: *Temperature* and *Critical level* output signals. Therefore, the following tuple is abstracted from the model: $\langle less\_than, (Temperature, Critical\_level)\rangle$.

It could also be described with only one signal (*Temperature*) and a constant (706). Such behavior-sequence pair may be used in different rules on the proposed OIU discussed for **Example 1**. Accordingly: in S_1, the oracle must analyze whether the pair holds for more than a given interval and, in S_2, it must analyze whether the safety protocol starts within an acceptable delay and if it lasts for a given interval.

Oftentimes, a rule may express relations between behavior-sequence pairs. For example, if a behavior-sequence A holds, then a behavior-sequence B must hold for some time.

Definition 15 - ***Implication rule*** *is a rule that can be written with an implies logical connective as* $P \implies Q$ *where* P *and* Q *contain at least one behavior-sequence pair each.*

A Simulink model may contain thousands of blocks and its respective oracle specification may be large. The modularization layer addresses the organization of OIUs into nodes of a tree structure to provide an hierarchical view and navigation of the oracle information.

Definition 16 - ***Modularization layer*** *is an abstraction which addresses the definition of a tree structure for organizing OIUs.*

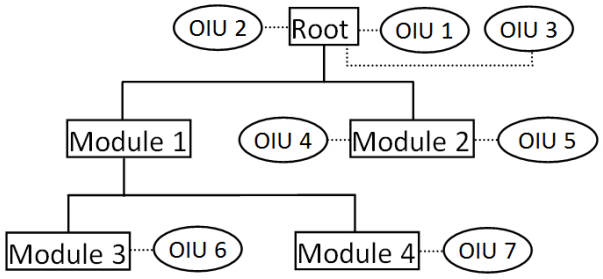


Figure 4.3: Modularization

Figure 4.3 shows an example of modularization with three levels. The first level is called *Root*. It contains three OIUs and two modules: *Module 1* and *Module 2*. The former has no OIUs but two modules, each one with an OIU. *Module 2* contains two OIUs. Other modules may be also defined in a tree hierarchy, as *Module 3* and *Module 4*.

As already discussed, the oracle information definition should not be restricted to the existence of a model. Four scenarios were considered: (i) the tester only has a Simulink model, (ii) there is a Simulink model and a high level specification; (iii) there is only a high level specification but no model; and, (iv) the project just started and no document or model is present.

In the first scenario, it may be natural that the modularization follows approximately the model subdivision structure – the model subsystems. For example, Figure 4.4 shows a possible modularization of the model from Figure 4.2. An OIU which references three subsystems (from **Example 1**) is defined in the *root* module. Three other modules were created to comprise OIUs which test requirements specifically related to the respective subsystems: *controller*, *indicator* and *safety protocol*.

But other modularization approaches may be applied. For example, if the tester already has a documentation and its requirements are already modularized, he/she may adopt such structure. Section 7.3 presents an experiment which contains a *real-world* documentation used as basis to the modularization, instead of a subsystem modularization approach.

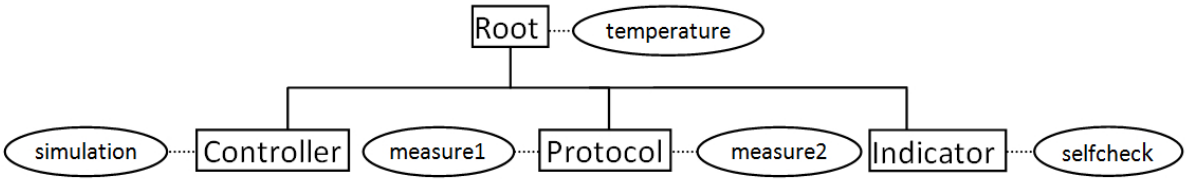


Figure 4.4: Modularization of Figure 4.2

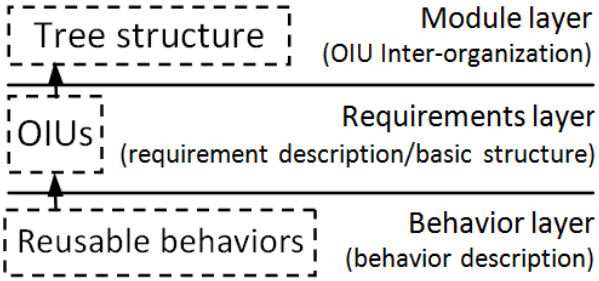


Figure 4.5: Oracle Information layers

In case no documentation or model exists, the development team may use the oracle generator as a framework of test driven development.

Figure 4.5 illustrates the layers and their interrelations. The requirements layer, the core of the oracle specification, represents the system requirements as Oracle Information Units. An OIU may refer to one or more behaviors defined in the behavior layer and can be reused in other units without being re-stated. OIUs can be also organized into a tree structure.

This elicitation effort can be softened by a partially automated oracle generator tool, as shown in Chapter 5.

4.3 Mapping

Mapping is the relation between a concrete data (signal) and its respective representation in the oracle information (attribute). It allows the oracle procedure to extract the simulation outputs and analyze them with respect to the oracle information. Relations may be expressed as a table with two columns representing, respectively, signals and attributes, in which each line describes a mapping.

Simulink-like tools (Chapter 2) have libraries with blocks which represent different types of outputs and inputs, respectively named as *sink* and *source* libraries. Examples of sink blocks are *to file* and *to workspace*. The former dumps the simulation data of a signal into a file. The latter stores the data into a variable which can be accessed in the Matlab interface (or other equivalent tool as Scilab and ScicosLab). But considering only these libraries of blocks would narrow the test activity. As such blocks are system (or

subsystem) interfaces, their use would allow only functional (black box) testing or system and a subset of integration testing.

However, if any signal of the model is subject to be mapped, it can increase the range of testing options.

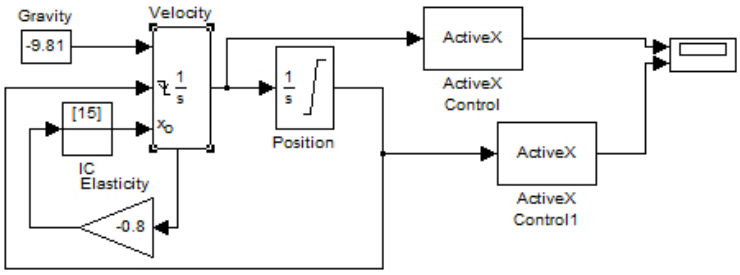


Figure 4.6: Mapping options. Source: (Mathworks, 2013a)

For example, Figure 4.6 presents a bouncing ball subsystem from MathWorks [1]. If the oracle only allows mapping between input and output blocks, the access to the model would be limited only to the display (rightmost block) and *Gravity* block (leftmost block). Otherwise, if any signal is mappable, the oracle may analyze the relation between any part of the model, including relations between subsystems. It may be analyzed, as instance, if *Velocity* block behaves correctly with relation to any of its three input signals and its output signal.

When a simulation is performed, a sequence of values is produced for each signal in the model and the oracle must be able to reference such values as an attribute in its analysis. At each analysis instant, a mapped attribute must assume a value from a sequence. For example, let us suppose that a simulation with 10 instants generates a sequence of 10 values uniformly distributed from 3 to 12 for signal *Velocity* and the tester wants to evaluate some simple requirement as *velocity must be less than 8*. In such case, he/she defines an attribute in the oracle information called *velocity*, maps it into the model and uses the attribute in a rule as $velocity < 8$. The oracle analyzer must then recognize that at instant 0, the mapped attribute *velocity* represents 3 and so on till instant 9, when the attribute represents 12.

Therefore, the values of each mapped signal produced by the simulation must be extracted by the oracle analyzer. The extraction proposed in this thesis is accomplished by instrumenting each mapped signal in the model with blocks that produce log files during the simulation. Each log file represents a mapped signal and named with a unique label wich is registered in the mapping table. It allows the oracle to retrieve the data.

[1] `http://www.mathworks.com/help/gauges/examples/bouncing-ball-subsystem.html`

A study presented in Section 7.2.2 indicates that such instrumentation does not affect the simulation performance.

4.4 Oracle Analysis

An oracle procedure compares the oracle information with the obtained result (Durrieu et al., 2008). As earlier discussed in Section 2.2.1, the access to the simulation outputs may be off-line or on-line, which may reflect in the oracle analysis, resource demand and on the sequence storage.

This section describes oracle procedures for both on-line and off-line sequence access. It also points a difficulty in analyzing rules for a specific condition called *close interval* and how it can be overcome by the procedure with oracle assumptions.

4.4.1 Off-line Oracle Access

If the sequence access is off-line, the simulation outputs are stored into some repository and the oracle retrieves them during its analysis. A benefit of such approach is that all the data is guaranteed to be available to the oracle. The oracle does not need to wait for a simulation instant, as it may happen if a rule expresses some temporal property. In this scenario, the analysis may require a value in the future (w.r.t. the analysis instant). For example, given a requirement:

If A is true then B must be true in the next instant.

An on-line oracle procedure would not be able to analyze such requirement at the first simulation instant where A is true, because the next instant is yet to be simulated. Therefore, the oracle would have to wait until the next instant to evaluate if the requirement is respected. However, an off-line oracle may search into the output repository to access the next instant.

Since the output data represents values of each mapped signal for each instant, a simulation may be composed by thousands of instants, producing the same number of values per signal. If the oracle analyzes all the simulation instants, it may read thousands of values of many sequences and the reading time cost may be a critical resource.

In this section, it is presented *Fast Jumper*, our proposed algorithm to read and write a sequence of values of repositories, as log files. It takes advantage of multiple references to a same sequence and benefits of shared memory spaces for temporal property analysis.

For illustration purpose, let $\langle A, a\rangle$, $\langle B, b\rangle$ and $\langle C, a\rangle$ be behavior-sequence pairs where: A is a behavior that must be observed in a sequence a (for example, if *pressure* is *greater*

than 10 at some point in the sequence); B is a behavior related to a sequence b; and C is related to sequence a.

And two abstractions of main rules:

(1) *If* $\langle A, a\rangle$ *holds,* $\langle B, b\rangle$ *must hold (for each instant)*

(2) *If* $\langle C, a\rangle$ *holds at some instant,* $\langle C, a\rangle$ *must hold again at least once in the next* X *instants, starting from the* Y *instant in the future*

First rule references two sequences, a and b, and second rule references only sequence a. Then, the same sequence (a) is referenced by two different rules. Also, the second rule has a temporal property: the behavior C must occur again at least once in the next X instants, starting from the Y^{th} instant in the future w.r.t. the analysis instant.

The algorithm starts the analysis at instant 0 and continues till the last simulation instant. At each analysis instant, it analyses all the existent rules.

Every sequence has a main memory space which stores a preset quantity Q of values from a sequence. Q may be computed automatically based on the available memory and the number of sequences mapped into the model. When the next analysis instant corresponds to a value beyond the memory space, it is repopulated by the next Q values in the sequence. A secondary memory space of size R, which may be attached to each behavior-sequence pair, is used when the rule has temporal operators that reference values beyond or before the main memory space boundary. R may be different of Q.

For example, let us consider $Q = 500$ values, $R = 100$ values, $Y = 600$ instants, $X = 1,000$ instants. For the sake of understanding, the second rule is rewritten as follows:

(2) *If* $\langle C, a\rangle$ *holds,* $\langle C, a\rangle$ *must hold at least once in the next* $1,000$ *instants, starting from the* 600 *instant in the future*

The procedure starts the analysis at $analysis\_instant = 0$ (the first analysis instant). At this moment, the first rule is analyzed. The first reading of a and b populates their respective memory spaces with the first 500 values of each sequence from the log files ($Q = 500$), as presented in the next code (lines 2 and 3):

```
1 retrieve_value_in_sequence(Attribute x){
2     if(current_MMS_index==Q || current_instant()==0){
3        MMS(x) = read_next_Q_values_from(log_file_x);
4        MMS_index=0;
5     }
6     return value_of(MMS(x),MMS_index);
7 }
```

MMS_index is a counter that represents the value in the memory space with respect to the current instant, that is, at each analysis instant, MMS_index is incremented by 1. The values are read and used in the comparison. Then, the second rule is analyzed. In this case, because a has already a memory space, the respective value is read only from the memory space, not the file (condition of line 2 is false).

When analyzing the second part of this rule, it must verify if the behavior occurs again in the future, between instants 600 ($analysis\_instant + Y$) and 1,000 ($analysis\_instant + X$).

Supposing that C holds at instants 0, 1 and 900, it means that the value of a at instant 900 is not in the main memory space, because its size is $Q = 500$. In this case, a secondary memory space is created and populated with values from instant 900 to 999 (or 851 to 950, considering 900 as the middle of the secondary memory space).

After all the rules are analyzed at instant 0, the *analysis_instant* is incremented by 1 ($analysis\_instant = 1$) and the analysis is executed again. At this instant, all the sequences are in memory spaces.

Next code presents a simplified algorithm for retrieving values not in the analysis instant.

```
retrieve_value(Attribute x, Instant i){
   if(current_MMS_index+i<Q && current_MMS_index+i>=0)
       return value_of(MMS(x),current_instant()+i);
   else if(current_SMS_index+i<R && current_SMS_index+i>=0)
       return value_of(SMS(x),current_instant()+i);
   else{
       SMS(x) = read_next_R_values_from(log_file_x,current_instant()+i);
       reset_SMS_index();
   }
       return value_of(SMS(x),current_instant());
}
```

First, the analyzer tries to retrieve the required value from the MMS. In case it is not in the MMS, it tries to retrieve the value from the SMS. If it is not in the SMS, R values are read from the log file into the SMS.

Analyzing all the rules for each instant, instead of analyzing one rule individually from instant 0 to the last instant then another one, allows a better memory space usage: all the rules that reference a same sequence share the same memory space. It prevents the algorithm to access the log file and the memory space repopulation more than once for the same analysis instant for interval of Q instants.

4.4.2 On-line Oracle Access

When the sequence access is on-line, the simulation outputs are sent directly to the oracle or it can be stored into a repository and made available to the oracle at runtime. A benefit of such access is that the analysis can be made *on the fly* and the tester does not need to wait for the simulation ending to have the oracle report.

However, an issue with on-line access is the data unavailability. If a rule contains temporal properties, as "in the next X instants in the future", the oracle may have to wait until the simulation reaches the X instants to access the required data and verify the rule.

The oracle may continue to analyze other rules while waiting for the unavailable data of a specific rule and put it on a waiting queue. When the data is available, the rule in the queue is verified. Thus, differently from the *Fast Jumper* algorithm, the rules are not analyzed all for each instant. As consequence, the rules in the queue may not take advantage of the shared main memory space if the waiting time exceeds the memory space size. Also, the oracle must track the instant where the rules were queued and all the data needed to verify the rule from the instant where it was put in the queue on.

A second solution is to stop the analysis until all the data is available, without a waiting queue. It allows the use of the shared memory spaces, but it drives to an idle analysis time.

In both cases, main memory may become a critical resource if the simulation output is sent directly to the oracle, without storing it first. The use of repositories even for on-line sequence access prevents such memory issue.

4.4.3 Oracle Assumption

Rules with temporal properties may be difficult to write. It is particularly true when these properties involve intervals of time.

Definition 17 - ***Interval of occurrences*** *is a finite sequence of instants $(i_h, i_{h+1}, ..., i_{h+l})$ w.r.t. a sequence s, where l is an integer greater or equals to 0, and a behavior b is true for all instants from (i_h to i_{h+l}), but not true at instants i_{h-1} (if i_h is not the first analysis instant. Otherwise, this last constraint is ignored) and not true at instant i_{h+l+1} (if i_{h+l+1} is not the last analysis instant. Otherwise, this last constraint is ignored).*

An interval of occurrences may be present multiple times in a sequence. For example, Figure 4.7 represents a simulation scenario with two behavior-sequence pairs, A and B. Black means that a pair holds at the respective instant. In this scenario, A holds at three intervals of occurrences: from instant 0 to 6, 9 to 16 and 18 to 21.

When analyzing implication rules as $A \implies B$, the oracle must relate the occurrences of a behavior-sequence pair A with the occurrences of other behavior-sequence pair B. However,

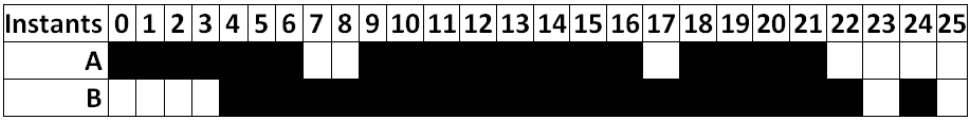


Figure 4.7: Close interval of occurrences

when multiple intervals of occurrences are too close each other, it may be impossible to the oracle to identify such relation without previous instructions.

For instance, the rule *"If A holds for an interval, B must hold for the same number of instants, with a maximum delay of 5 instants"* could be easily described with temporal logic or timed automata if such behavior relation is expected just once or on distances greater than 5 instants between intervals of occurrences. But in the scenario of Figure 4.7, it is impossible to decide what is the delay of B in relation to the second interval of occurrences of A.

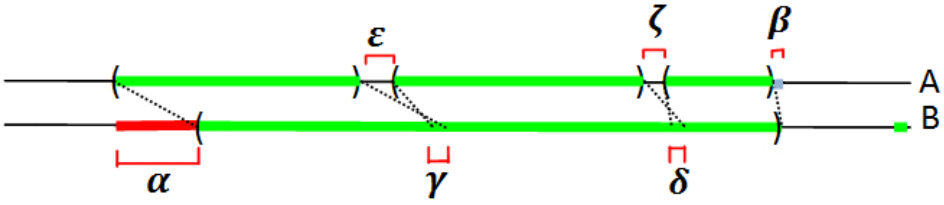


Figure 4.8: Undecidable relation between A and B

Figure 4.8 helps to illustrate the undecidability of the same scenario: interval α represents a delay before B starts holding in relation to A and it is possible to define the size of the subinterval of B which may be related to the first interval of occurrence of A. But it is impossible to define with precision in which instant B starts to hold in relation to the second occurrence of A. Therefore, it is impossible, in the absence of some assumption, to define the size of γ. In the same way, it is not possible to define the size of δ.

The oracle may consider that instants 4 to 10 of B are related with the first interval of occurrences of A and that instants 11 to 18 of B are related with the second interval of occurrences of A. Or, the oracle could consider the same with the first relation between B and A, but that instants 9 to 16 of B are related with the second interval of occurrences of A.

Such difference of assumptions may interfere in the oracle final report. With the former (**successive assumption**), it implies that the scenario agrees with the rule, but at instant 24, B is not related with any occurrence of A (here called **spurious occurrence**). With the latter (**overlapping assumption**), B would not be related with A at instants 17, 22 and 24. In other scenarios, the rule verdict could even vary from agreement to failure.

Definition 18 - *When intervals of occurrences are too close so that ambiguous analysis interpretation may be present, it is called* ***close intervals****.*

Often a simple natural language description may be hard to formalize unambiguously for such scenario and analysis assumptions must be made to reduce the formalization complexity.

Spurious occurrences and the relation between intervals of occurrences are bound with the way in which the oracle performs its analysis.

Definition 19 - *An **oracle assumption** is an admissible inference that the oracle adopts during the analysis in the presence of close intervals. It aims to reduce the rule formalization complexity when close intervals are expected.*

Definition 20 - *When the oracle (i) analyzes an implication rule ($A \implies B$), (ii) relates an analyzed instant of B with an interval of occurrences of A and (iii) infers that such instant must be ignored when searching for the next relation between B and a subsequent interval of occurrences of A, it is called a **successive assumption**.*

Definition 21 - *When the oracle (i) analyzes an implication rule ($A \implies B$), (ii) relates an analyzed instant of B with an interval of occurrences of A and (iii) infers that already related instants of B can also be related to a subsequent interval of occurrences of A, it is called an **overlapping assumption**.*

4.5 Final Remarks

This chapter describes an oracle engineering foundation to the partially-automated generation of test oracles for Simulink-like models. The proposed approach is intended to contribute with the test by providing a method to define an oracle information, mapping between model and information with a proposal of instrumentation and an oracle procedure which analyzes the data retrieved from a simulation w.r.t., possibly, temporal specifications. It also discusses solutions for two main concerns: data access and assumptions over close intervals. Finally, 21 definitions were listed in order to facilitate the understanding of the approach.

Many steps are automatable: (i.a) instrumentation may be accomplished automatically during the mapping; (ii.a) simulation data recovery; (iii.a) the analysis and report may be automated if the oracle information is expressed with enough formalism.

Manual efforts may be softened by support tools: (i.b) editors with high usability which reproduce the process steps with simple options; (ii.b) wizards to compose the rules; (iii.b) graphical representation of a model to support the mapping with the oracle information.

Next chapter presents Apolom, a tool implemented with the concepts here discussed, which allow the definition of oracle information based on specification-languages with capacity to express temporal properties.

Chapter

5

Automating an Oracle Generator

Previous chapter presented the fundamental foundation concepts of Simulink-like oracle generation, enabler of oracle automation.

This chapter describes Apolom, an oracle generator tool that provides support to all three activities of the oracle definition process (Chapter 4): the specification definition, the mapping between specification and model, the configuration and simulation analysis.

Its oracle information and analyzer were implemented upon an specification-based language, TRIO, due to its expressiveness and usability.

Apolom is an important contribution in two ways: it represents an instance of our approach, that is, a demonstration that an oracle generator tool is feasible; and it is a vessel to the evaluation study of the proposed solution (Chapter 7).

Section 5.1 presents the main features of Apolom highlighting the automated steps and how manual efforts were softened by the tool. Section 5.2 describes its functionalities and graphical interfaces with a running example. Section 5.3 discusses the impact of different oracle assumptions in the analysis result. Section 5.4 presents the trigger relation between rules within an OIU. Section 5.5 concludes the chapter. Next chapter discusses implementation and limitations of Apolom.

5.1 Features

The tool partially automates the oracle generation. The **manual effort** of the process includes the specification writing and the mapping between a Simulink model and the specification. For both steps, the tool provides means to alleviate the load over the tester.

Specification writing: this step is softened by editors disposed in a sequence that establish a straight line through the generation process described in Section 4.1, starting at the attribute definition and mapping, and passing by the module, behavior and OIU definitions.

Apolom also provides a Rule Wizard composed by two parts related by the propositional connector *implies* (Figure 5.1(A)). In each side, it enables the definition of expressions that embed the following triple: attributes (Figure 5.1(B)), behaviors (Figure 5.1(C)) and shortcuts (Figure 5.1(D)).

Shortcuts are definitions commonly used in the base language (TRIO), for instance, *during an interval* or *with a maximum delay of*.

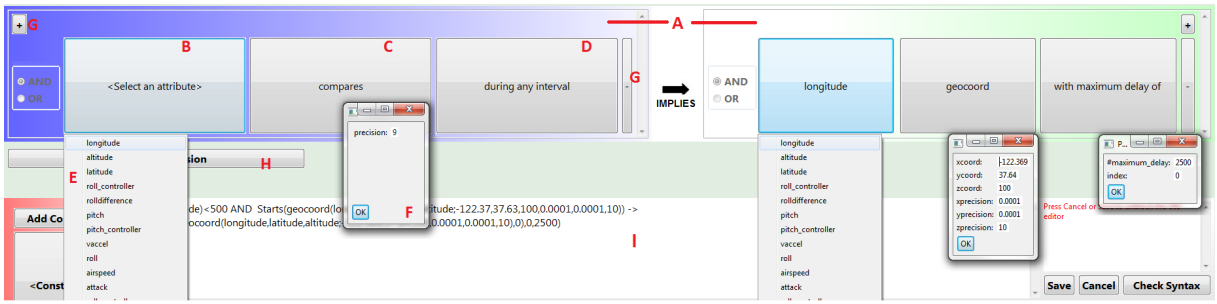


Figure 5.1: Rule Wizard

After an attribute is defined, it is automatically listed (Figure 5.1(E)) by pressing the respective component. The same occurs with the existing behaviors and shortcuts. In these cases, they may require a parameter configuration. As instance, the following behavior can be defined in the behavior editor:

$$compares(seq1, seq2; precision) =$$
$$[Current(seq1) >= Current(seq2) - precision \text{ AND}$$
$$Current(seq1) <= Current(seq2) + precision]$$

It holds when a value from a sequence is equals to a value from another one, within a margin of acceptance, at a given analysis instant. This margin is represented by a parameter (*precision*), which is automatically presented in a window (as in Figure 5.1(F)) when its respective behavior is selected.

Such a behavior may be particularly useful on consistency checking, in which points in the model must be verified in relation to design aspects which transcend the functional

requirements of a system. A real-world example is given in Section 7.3.2: a consistency requirement states that a lever smoothing roll value from an airplane is the same in the Pilot subsystem and in the Roll Controller subsystem, with a maximum tolerance of 2 degrees.

More triples $\langle attribute, behavior, shortcut \rangle$ may be added or removed by pressing the respective buttons (Figure 5.1(G)). Once the rule wizard is set, the expression is generated by pressing the *generate* expression button (Figure 5.1(H)).

Mapping: this step is softened by an editor that presents the diagram of the model and the list of attributes of the specification. It allows the tester to select the line to be mapped to an attribute directly from its diagram.

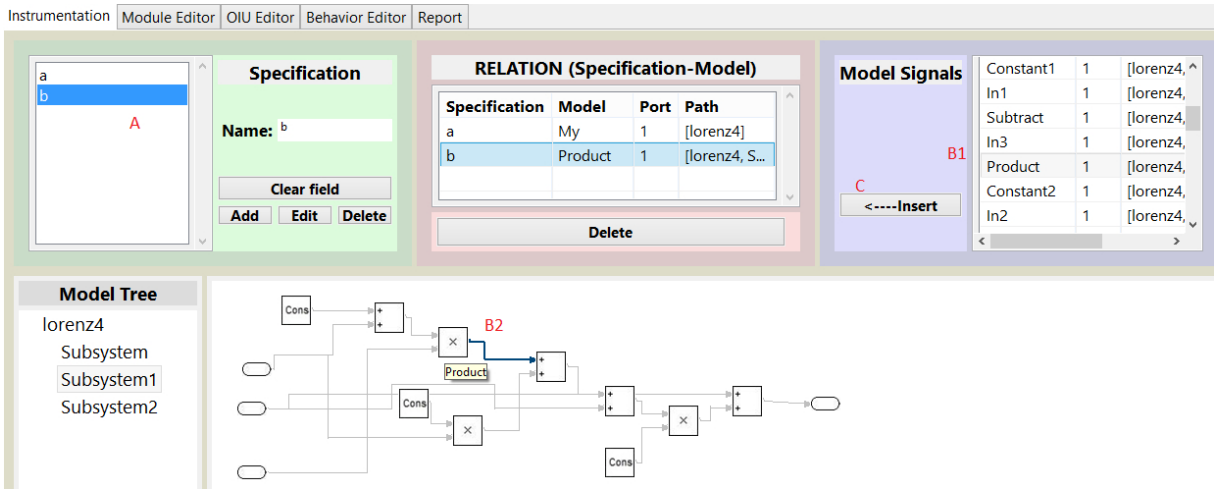


Figure 5.2: Mapping sample

Figure 5.2 shows an example of mapping. The attributes are listed in the *Specification* box (upper left corner), from where one may define, delete, edit or select the respective attribute to be mapped. The model lines (also called signals) are listed in the *Model Signals* box (upper right corner) and presented in the center of the screen. The tester can select the line to be mapped from both its list (Figure 5.2(B1)) or diagram (Figure 5.2(B2)). With attribute and line selected, the mapping is accomplished by pressing the *Insert* button (Figure 5.2(C)).

The **automatic steps** of the oracle generation process include the model instrumentation, trace recovery and oracle analysis.

Model instrumentation: for each mapped line, the oracle parses the original file of the Simulink model and inserts a dumper block (Figure 5.3, circled in red) into a copy file – the instrumented model. A dumper block is connected to a mapped line and sends the line trace into a log file when the instrumented model is executed.

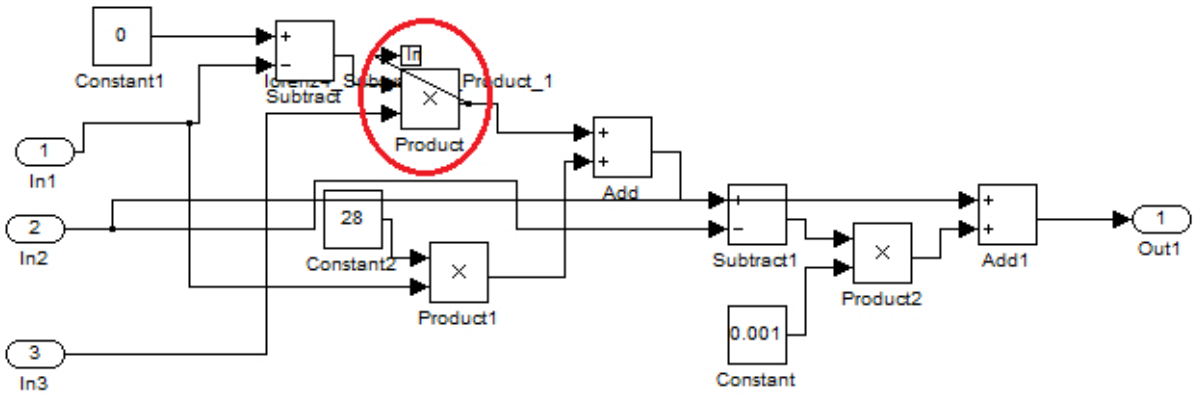


Figure 5.3: Instrumented model

Dumpers are subsystems composed by a *Data Type Conversion* block and a *To File* block. The former converts the input type into a double. The latter dumps the line trace into a file with unique name within the project folder.

Log recovery: at the end of the instrumented model simulation, each instrumented line will generate a log file. If an attribute must be read during the oracle analysis, its value is recovered from the log.

Oracle analysis: for each instant, the oracle analyzes all OIUs and reports whether it is violated, following the violation level of Section 4.2.2: critical failure, constraint failure, constraint critical failure or non-critical failure. It also reports which rules were disrespected and the values of each related attributes. The procedure also stores the state of the oracle during the analysis in a way that, if the information is changed, the report will not present out-of-date, and inconsistent, results. The oracle analysis is described in details in Section 4.4 (concepts) and Section 6.3 (implementation).

Next sections elucidate Apolom features, as its application with a running example.

5.2 Running Example

This section presents complete example of an oracle generation with Apolom, intended to provide a clear overview of the previously described functionalities and operation. It discusses each step of the oracle process and how it was adapted to the implementation.

The model under test was based on the scenario of Figure 4.7, in Section 4.4.3. It simulates a boiler controller in which two lines must be inspected: *pressure alarm* and *pressure controller*. The first line represents an alarm that has a true value when the input pressure is above some preset limit or false otherwise. The second line represents the controller warning and is true when the control is on. The boolean values are represented by 0 or 1.

The requirement states that, if pressure alarm is on, then pressure controller must be on within the next 3 instants for at least the same instants as the alarm.

The oracle project starts by selecting a project name and folder. The model may be loaded later in the process (Figure 5.4).

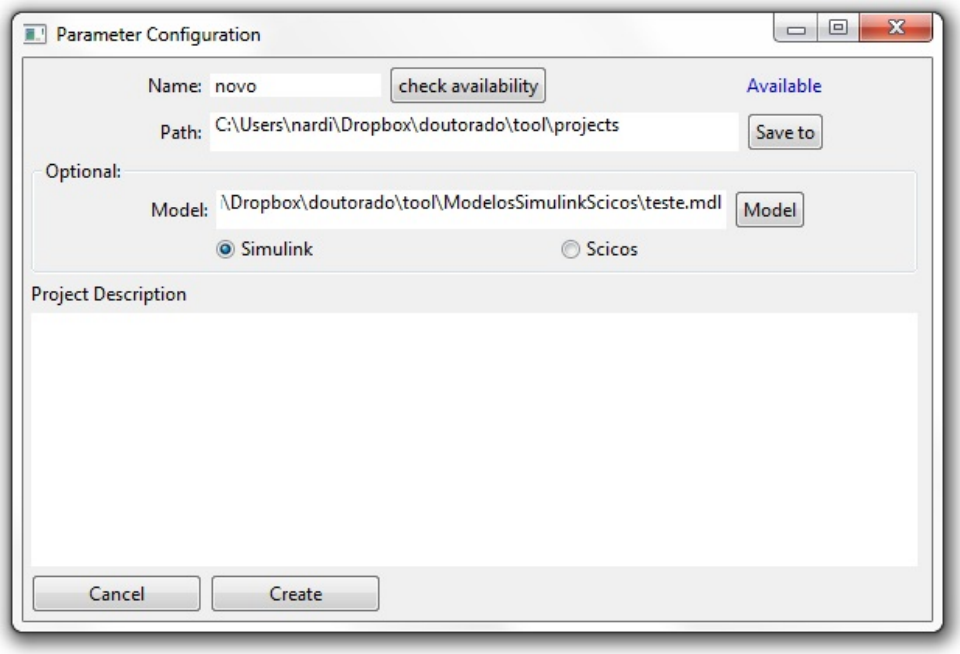


Figure 5.4: Model and folder selection

The oracle definition follows three steps (Chapter 4): information definition, mapping, and analysis configuration. The information definition relies on three layers, as described in Section 4.2.2. In the tool, the layers are designed in the respective tabs – module, OIU and behavior.

Next subsections present each step in details and the analysis execution.

5.2.1 Attribute Definition and Mapping

The mapping is a step that can be performed at any time after the specification attributes are defined. It allows the tester to write the oracle specification without the presence of a model, as discussed in the end of Section 4.1.

The *Instrumentation* tab (Figure 5.5 (A)) allows both actions: attribute definition and mapping.

The attribute is defined by providing a name and adding it in the appropriated section (Figure 5.5 (B) and (C)).

The mapping between an attribute and a line from the model is performed by selecting the line from a list or model diagram (Figure 5.5 (D)), the respective attribute and by

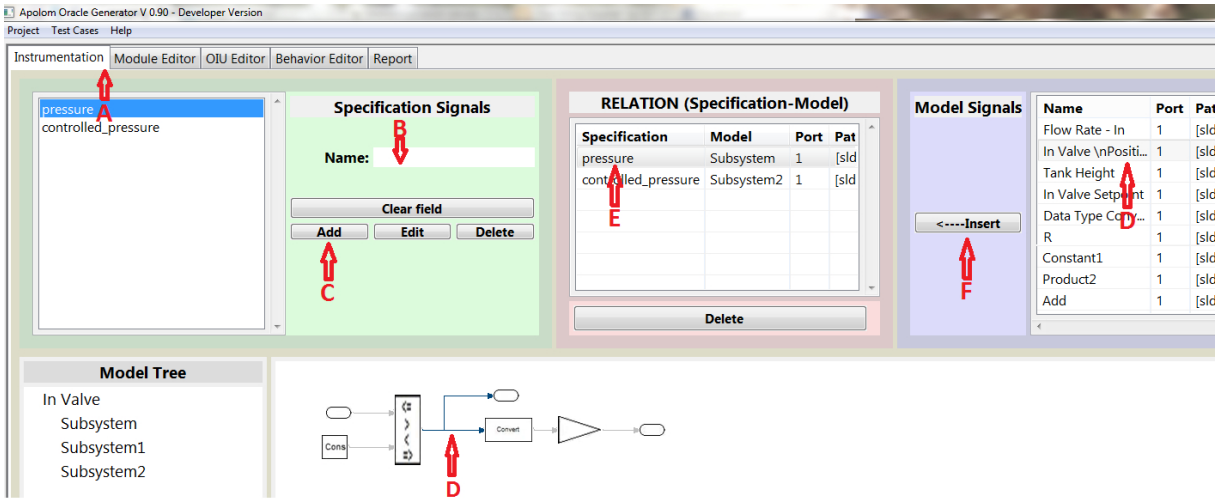


Figure 5.5: Intrumentation editor

pressing *Insert* (Figure 5.5 (F)). The relation is presented in a table in the upper-middle of the screen (Figure 5.5 (E)). The model instrumentation is automatically executed during the mapping.

5.2.2 Module Layer

Once the mapping is accomplished (or at least, attributes are defined), the tester may move to the *Module Editor* tab (Figure 5.6 (A)) and define the module structure in the bottom box, as proposed in Section 4.2.2. The first module in the tree structure is called *root*. From this node, the tester may create other modules, submodules and OIUs.

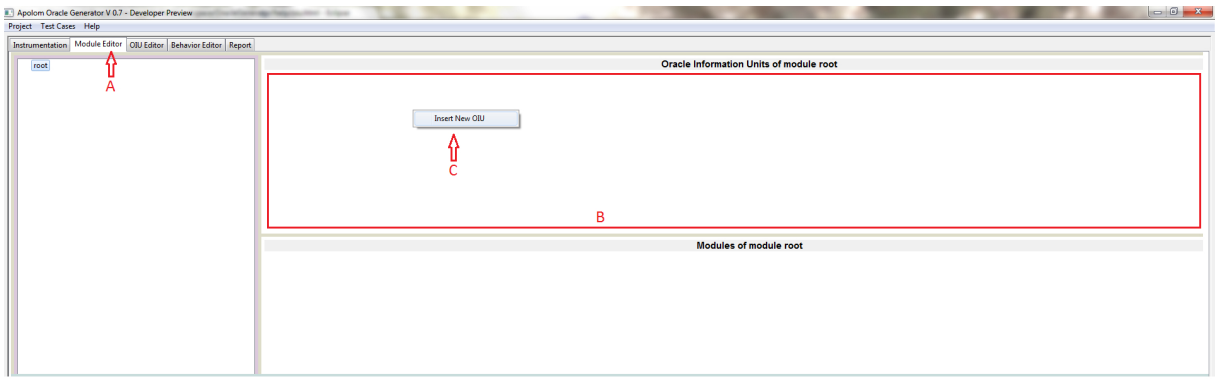


Figure 5.6: Module editor

Given the simplicity of the example, which contains only one requirement, there will be no modules, except for the root. The OIU representing the requirement must be defined within its respective module. The tester selects the correct module and adds an OIU into it by right-clicking in the proper box (Figure 5.6 (B)) and selecting a suitable name (Figure 5.6 (C)).

An OIU is represented as shown in Figure 5.7.

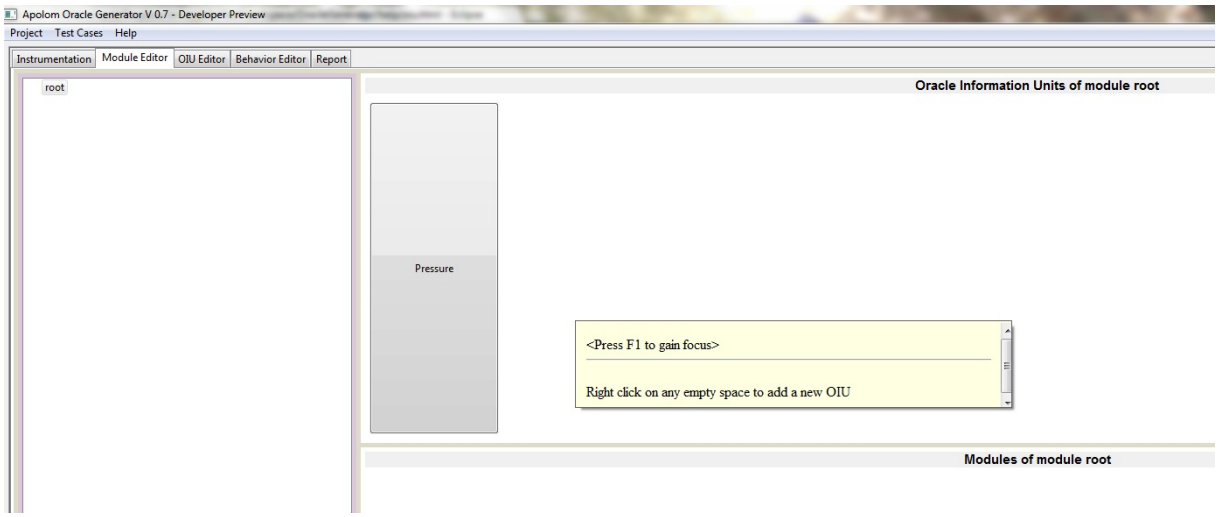


Figure 5.7: A defined OIU

The tester can, then, access the *OIU editor* (Figure 5.8) by pressing the respective OIU representation.

5.2.3 OIU Layer

An OIU is composed by a main rule and, possibly, constraints and safeguards (Definition 8, page 47). Rules, in Apolom, are described as TRIO expressions. The language is here called TRIO/Apolom given the adaptations and limitations of the implementation (as no support to recursion).

A Rule Wizard was implemented to facilitate the tester work in defining TRIO expressions. As stated in Section 5.1, it represents an expression with triples $\langle attribute, behavior, shortcut \rangle$.

The tester starts with the selection of the rule type to be generated – the main rule, constraint or safeguard – by pressing the respective button (in the example, Figure 5.8 (A)). Secondly, the first triple can be defined starting by selecting the sequence (here called signal), as presented in Figure 5.8 (B) and (C)). The defined attributes are automatically listed.

The second component from the triple can be selected in the same way as the line (Figure 5.9 (A)).

All the available behaviors are automatically listed, as shown in Figure 5.9 (B).

In the example, the selected behavior, *equals to*, has a parameter called *value*:

$$equalsto(seq;value) = current(seq)==value$$

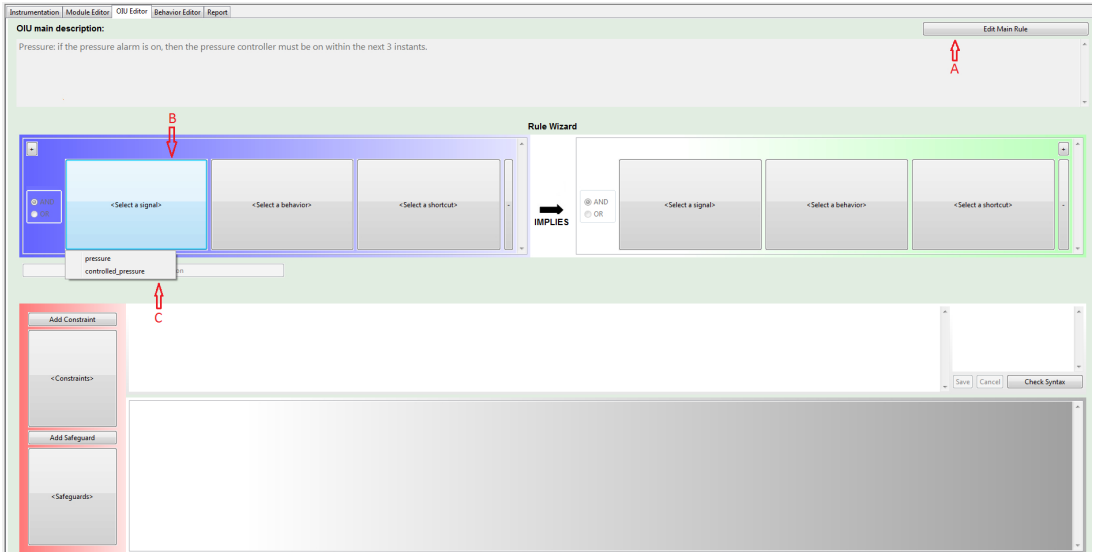


Figure 5.8: OIU editor

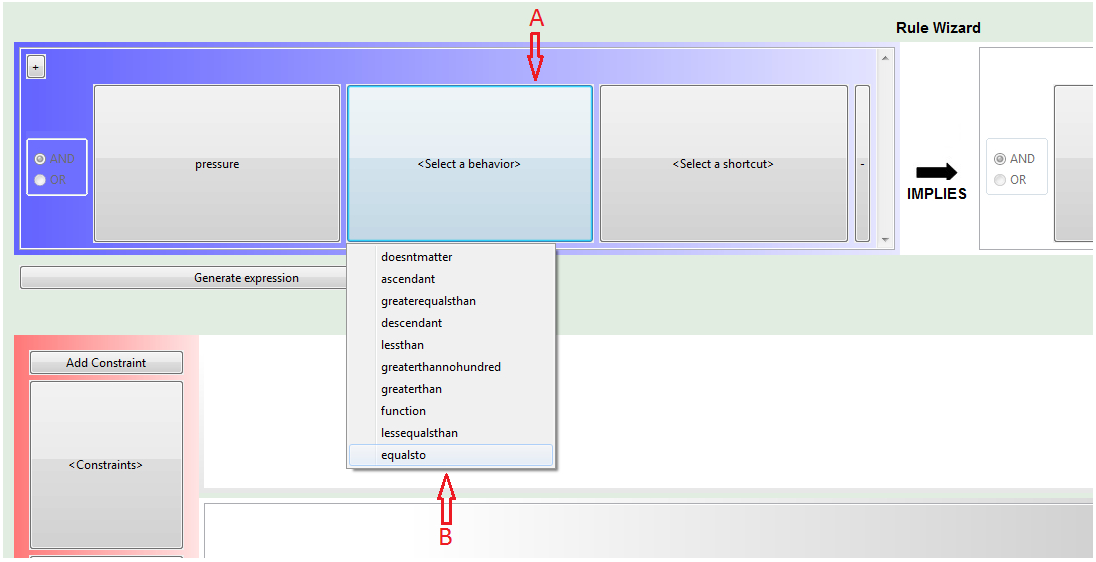


Figure 5.9: Behavior selection

In such case, the parameters are presented in a box (Figure 5.10 (A)). All parameter types are considered as double, although it is possible to adjust the maximum precision when comparing values.

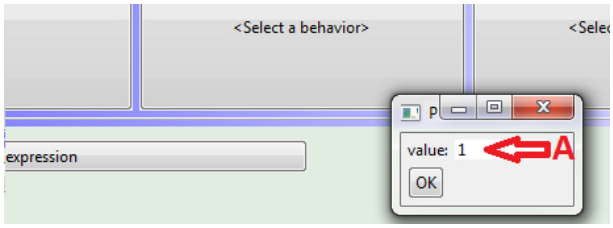


Figure 5.10: Parameter settings

The last component of the triple, the *shortcut*, is selected as the previous ones (Figure 5.11 (A)).

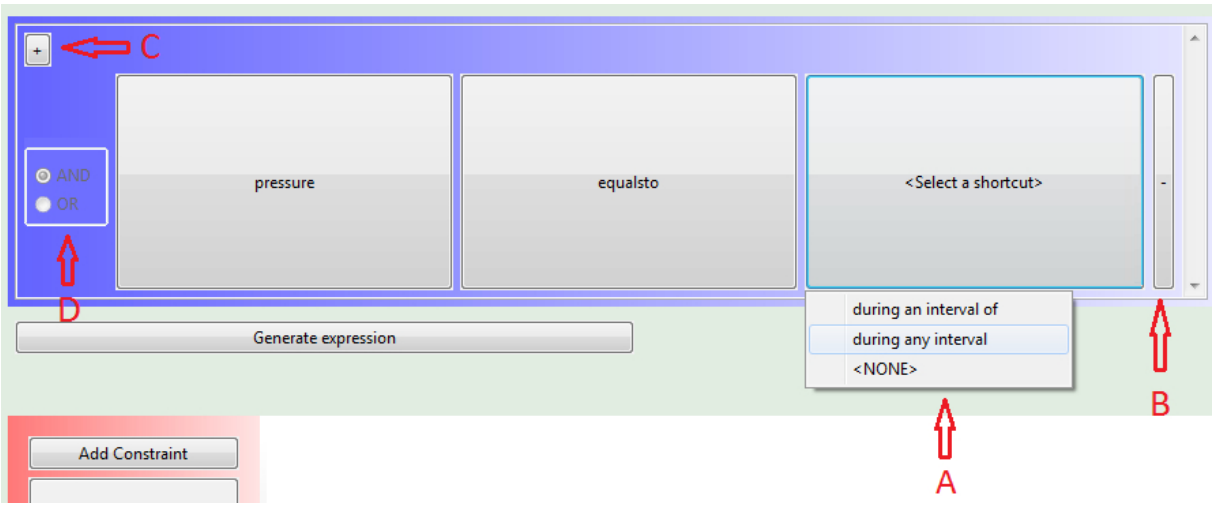


Figure 5.11: Shortcut selection

The same concept may be applied in the right side of the rule. It is important to note that the wizard does not generate only a rule with the format of *tripleA implies tripleB*. It is possible to define rules without the *implies* connector by removing the triple from one side (Figure 5.11 (B)). It is also possible to include more triples at any side (Figure 5.11 (C)) connected with other triples by *OR* and *AND* operators (Figure 5.11 (D)).

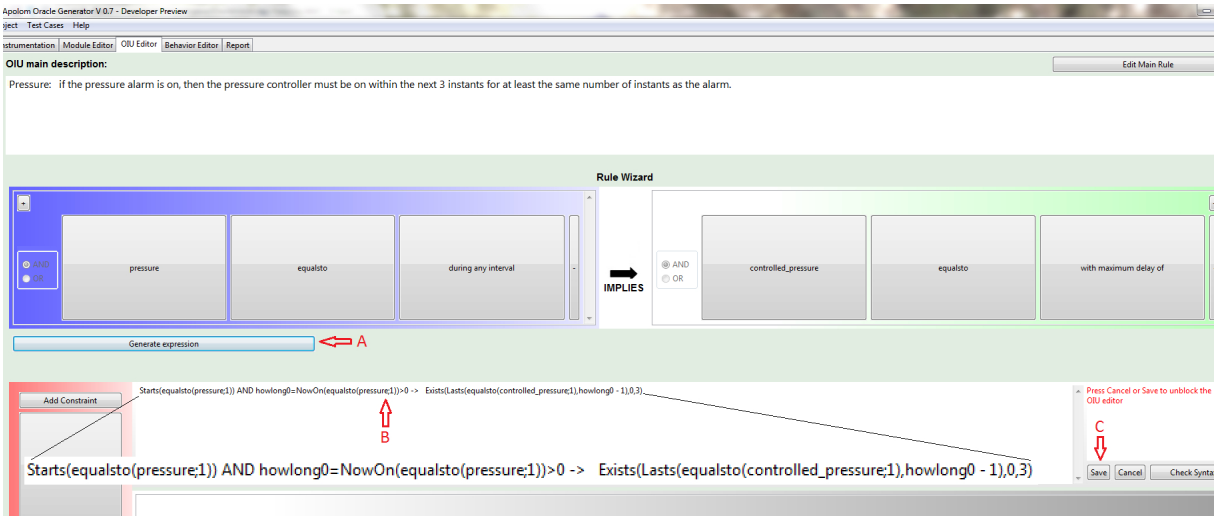


Figure 5.12: Rule generation

By pressing the *Generate expression* button (Figure 5.12 (A)), the rule is created (Figure 5.12 (B)). The tester may apply changes to the expression before saving it (Figure 5.12 (C)).

5.2.4 Behavior Layer

The behavior from the example is trivial and provided by the tool. However, the tester may define a new behavior in its respective tab (Figure 5.13 (A)). The tool shows two kinds of behavior writing modes: expressions or Java code (Figure 5.13 (B)), although the latter is not yet supported. The tool was also designed to allow the addition of different specification languages in the future, which will be selectable from the menu showed in Figure 5.13 (C).

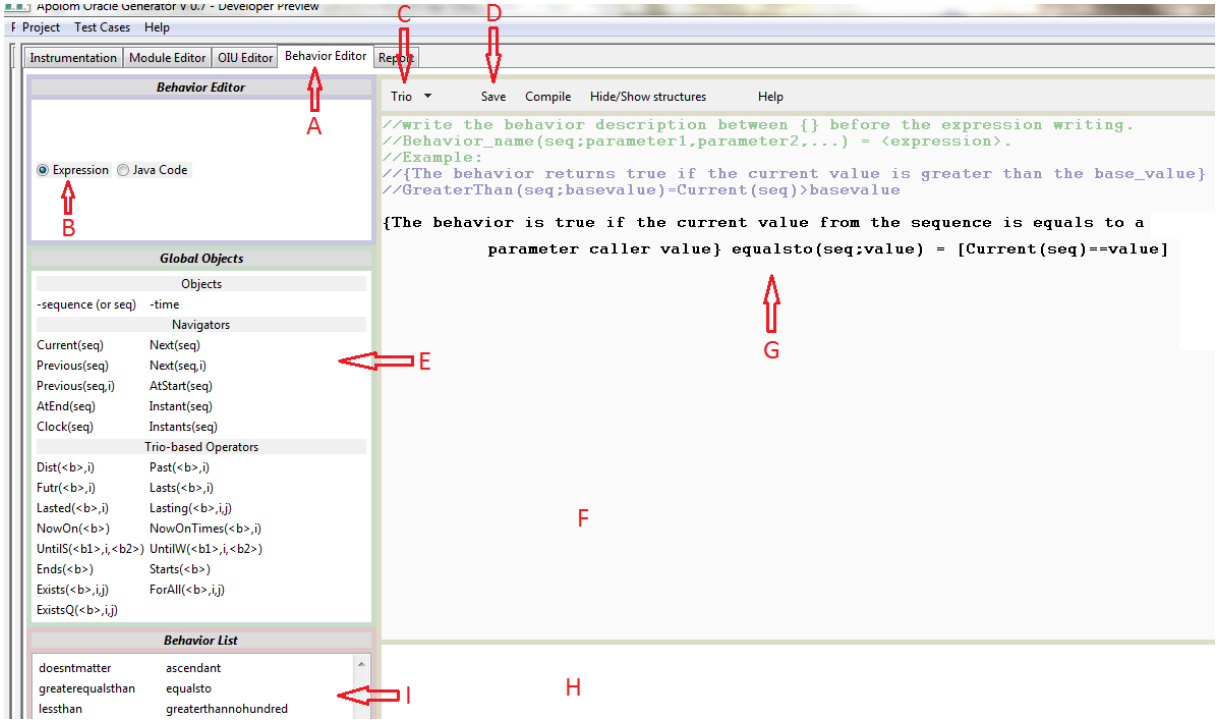


Figure 5.13: Behavior definition

To alleviate the burden of the writing, the editor presents the list of objects, navigators and trio-based operators (Figure 5.13 (E)).

The tester may write the behavior in the field showed in Figure 5.13 (F). Comments in green and blue present a description of the behavior's format. The behavior must contain a signature with a name and two types of parameters, separated by a semicolon: the sequences in the behavior and variables (Figure 5.13 (G)). It may also contain a comment between braces.

If more than one sequences or variables are necessary, they are separated by a comma. It is important to note that the parameter name of a sequence is not related with the real name of a sequence in a way that the same behavior may be used with different sequences. The relation between behavior and sequence is accomplished automatically by the Rule

Wizard, or manually in the rule editor. In the expression in Figure 5.12, sequence *pressure* is the argument related with parameter *seq* of Figure 5.13.

When the tester attempts to save a behavior (Figure 5.13 (D)), it is parsed and, in case of errors, they are presented in the field of Figure 5.13 (H). The behavior is added to the appropriated list (Figure 5.13 (I)) whether it is correct.

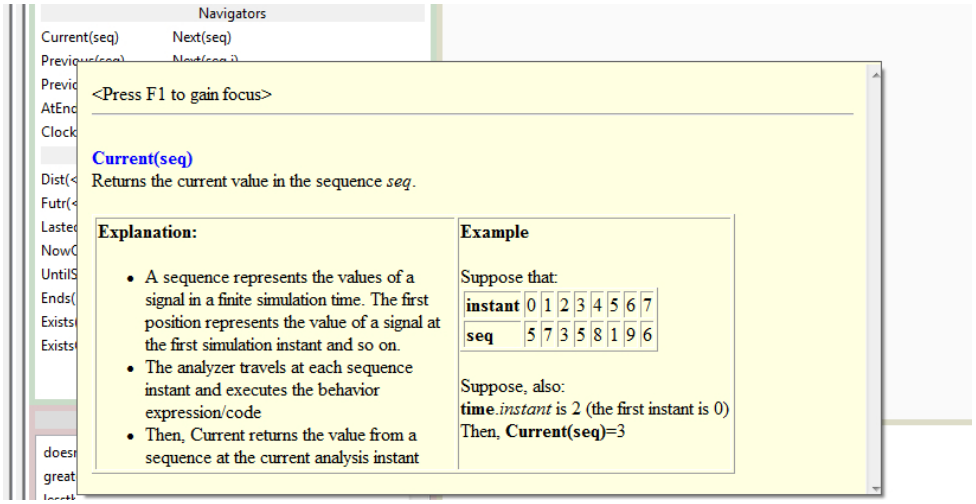


Figure 5.14: A tooltip

Apolom also provides tooltips for all behaviors, navigators and operators. The description made within braces in a behavior definition (Figure 5.13 (G)) is presented as tooltip. For navigators and operators, the tooltips are built-in. Figure 5.14 shows an example of tooltip for the *Current* navigator. It contains a brief description, a detailed explanation and an example.

5.2.5 Analysis Execution

When the oracle information is defined, the tester may configure the analysis execution (Figure 5.15 (A)).

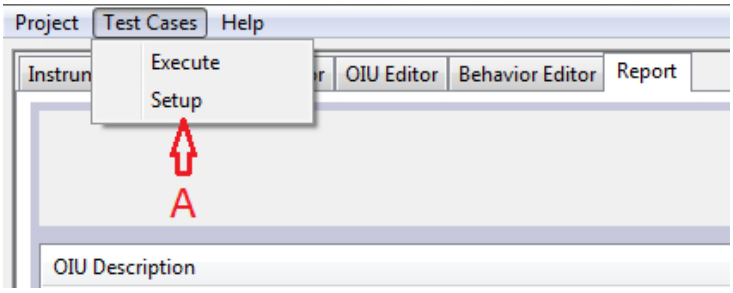


Figure 5.15: Oracle setup

The configuration includes the max number of instants to be analyzed (Figure 5.16 (A)), failures (B), analysis of spurious occurrences (C) and the analysis assumption (D) (Section 4.4.3).

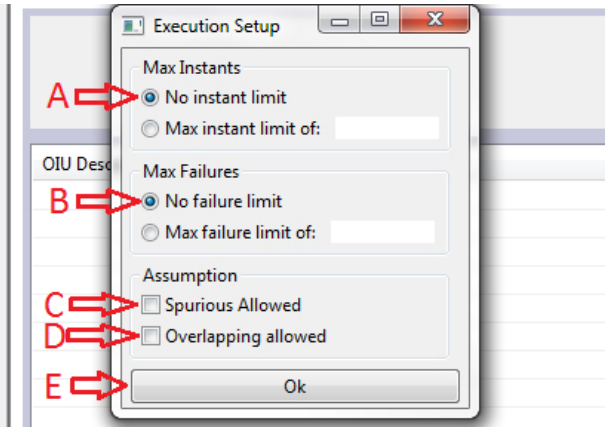


Figure 5.16: The configuration

The analysis execution produces a report which results may depend on the oracle assumption. In the first execution, the oracle must identify spurious values and adopt successive assumption (Definition 20, page 58).

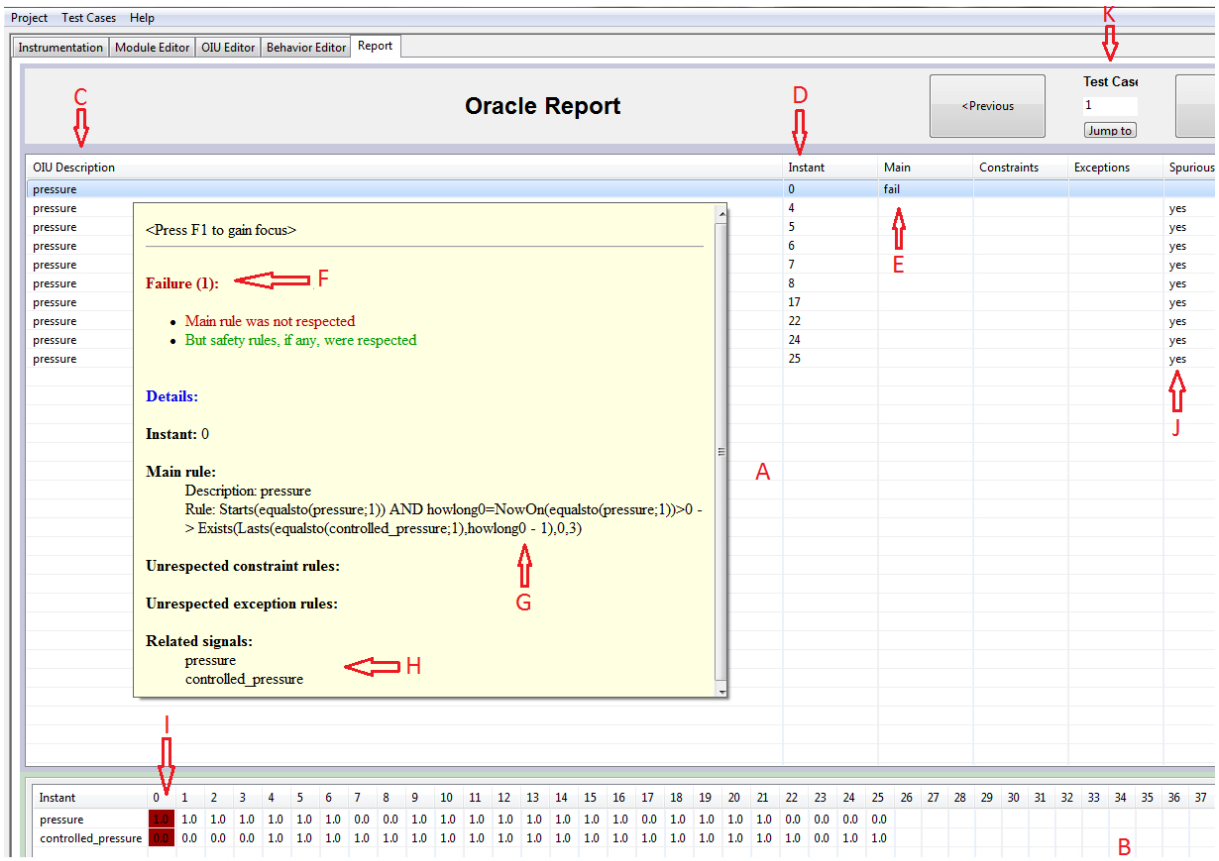


Figure 5.17: Oracle report

The main table from the oracle report (Figure 5.17 (A)) contains the description of the failed OIU (C), the instant in which the OIU failed (D), and which rule has failed. In the example, main rule failed at instant 0 (E). Selecting the respective line, a tooltip is shown with the failure type (F), which rules were disrespected (G) and the sequences related to the failed rules (H).

The sequence table (Figure 5.17 (B)) indicates the values of the involved sequences at the instant when the failure occurred (I) and the values in the vicinity interval. It helps the tester to identify in which instant and why a rule failed. For example, the first table presents a failure at instant 0 and the disrespected rule. Selecting the instant, the second table shows that at instant 0, in the first sequence, the value of *pressure* is 1. According to the rule, it should be a time in the next 3 instants when *controlled_pressure* should be 1. Analyzing the second sequence one can observe that it takes one more instant than the expected to the controlled pressure turns on (when its value is 1).

Spurious occurrences (J) were found at instants 4, 5, 6, 7, 8, 17, 22, 24 and 25. Because the first interval of occurrences of *pressure* (Definition 17) was not related with *controlled_pressure*, the occurrences of the latter from instants 4 to 8 were not related to any occurrence of the former.

5.3 Assumptions may Impact the Results

Different oracle assumptions are proposed to reduce the formalization complexity of temporal rules in a particular scenario called *close intervals*, as discussed in Section 4.4.3, page 56. This section presents an example of how different oracle assumptions may impact in the final report.

A slight change in the previous requirement helps to illustrate such impact: the pressure controller must be on within the next 4 instants, and not 3, as previously required.

The result is displayed as in Figure 5.18, still considering **successive assumption**, as explained next:

- First interval of occurrences from 0 to 6 of *pressure* will be related with interval from 4 to 10 of *controlled_pressure*;
- Second interval of occurrences from 9 to 16 of *pressure* will be related with interval from 11 to 18 of *controlled_pressure*;
- Third interval of occurrences from 18 to 21 of *pressure* will be related with interval from 19 to 22 of *controlled_pressure*;

- The occurrences of *controlled_pressure* at instants 24 and 25 will not be related with any occurrences of *pressure*, that is, they are listed as spurious values.

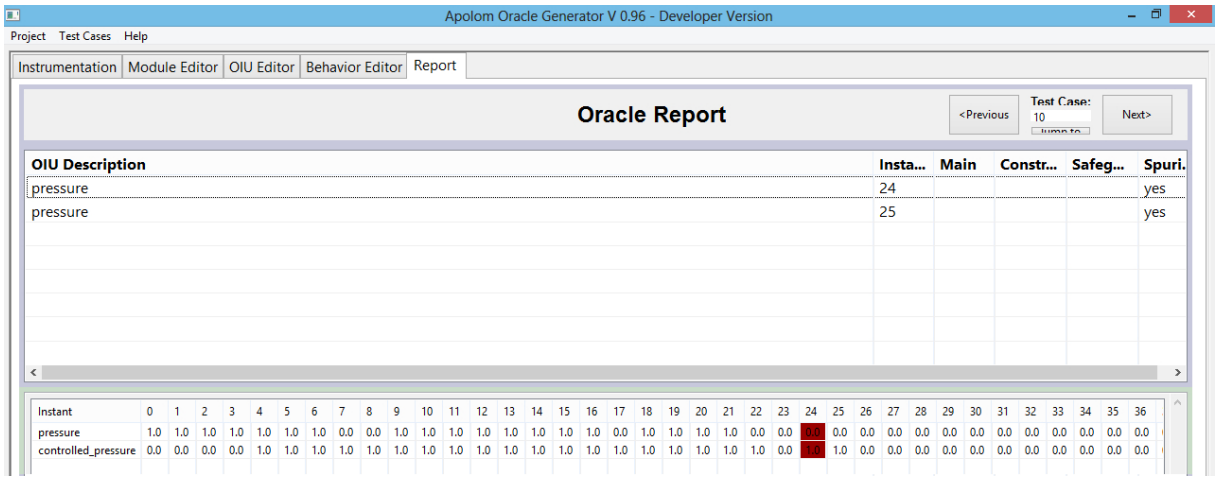


Figure 5.18: Oracle report: successive assumption

However, adopting **overlappig assumption** (Definition 21), the report will be different, as shown in Figure 5.19.

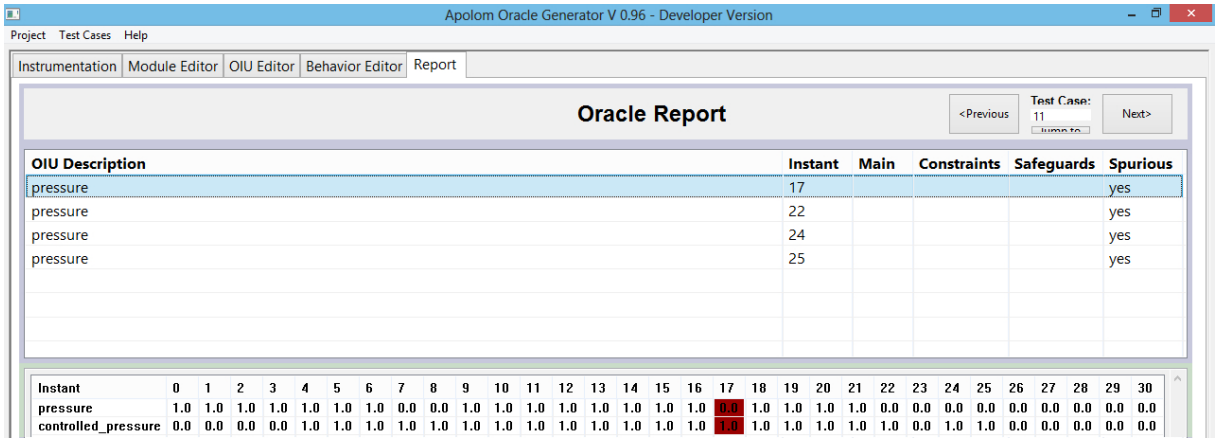


Figure 5.19: Oracle report: overlapping assumption

In this case:

- First interval of occurrences of *pressure* will be related with interval from 4 to 10 of *controlled_pressure*;
- Second interval of occurrences from 9 to 16 of *pressure* will be related with interval from 9 to 16 of *controlled_pressure*;
- Third interval of occurrences from 18 to 21 of *pressure* will be related with interval from 18 to 21 of *controlled_pressure*;

- The occurrences of *controlled_pressure* at instants 17, 22, 24 and 25 will not be related with any occurrences of *pressure*, that is, they are listed as spurious values.

In both examples, the oracle reports are accurate. However, they reflect different assumptions which impacts in the analysis mechanisms, possibly causing different results, as illustrated.

5.4 OIU: Trigger-Dependent Rules

This section overviews the dependency mechanism between an OIU main rule, constraints and safeguards.

A new constraint and safeguard was added to the already defined OIU from the last example. The constraint states that *alarm* must be off when *pressure* is under a critical level. And a safeguard states that if *pressure* is above the critical level for more than 8 instants and *pressure is not controlled*, then a safety protocol must be triggered.

First, a fault was inserted in the model so that the alarm remains on for two instants after the second interval of occurrences in which *pressure* is above critical.

Figure 5.20 shows that a constraint was disrespected in both instants (21 and 22). Selecting an instant, it also presents the failure type, the values of the related attributes and which rule was disrespected.

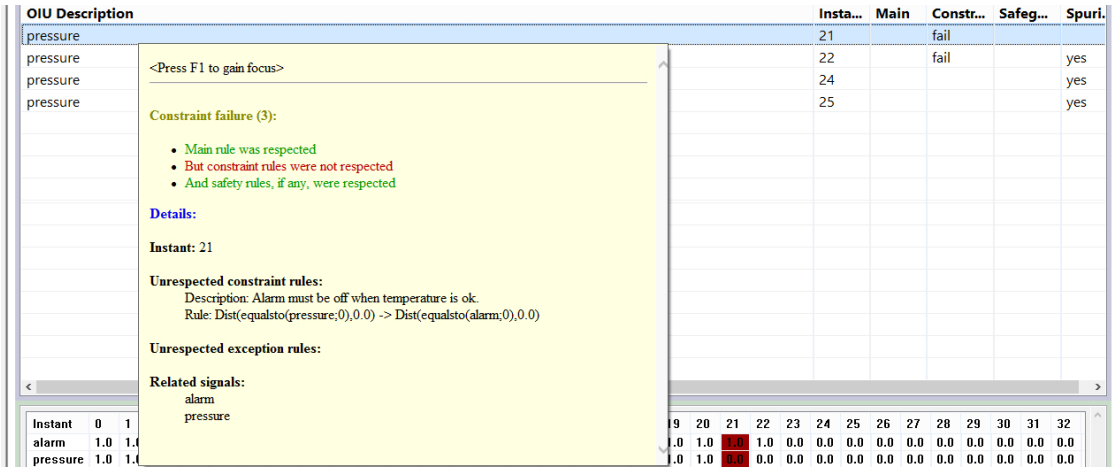


Figure 5.20: Oracle report: constraint failure

To present critical failures in an OIU, a fault was inserted in the safeguard subsystem in which the protocol is not triggered after *pressure* is above the critical for more than 8 instants. Another fault was also inserted in the model, so the pressure is not controlled in relation to the second interval of occurrences of *pressure* causing a main rule failure.

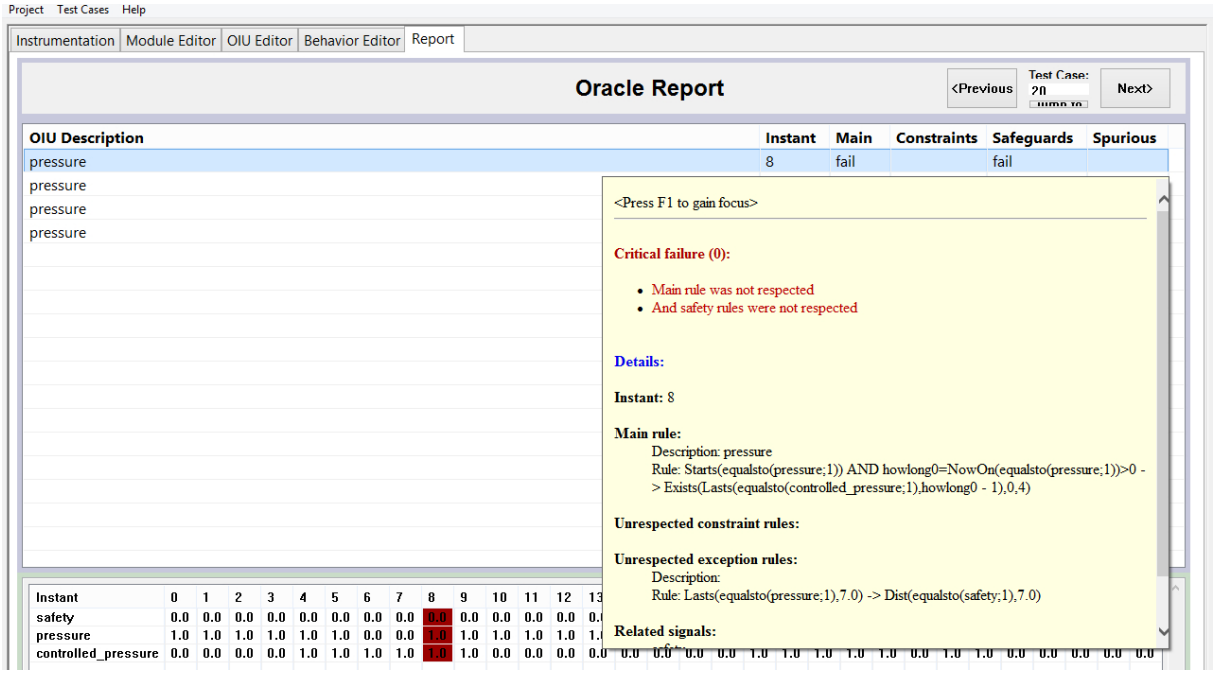


Figure 5.21: Oracle report: critical failure

Figure 5.21 shows the critical failure. It reports that the main rule and a safety protocol have failed. Additionally, three sequences are related with the failure: *pressure*, *controlled_pressure* and *safety*.

In the given scenario, it is not possible that constraint and safeguard are disrespected at the same time if the main rule is respected. It is true because this constraint is only checked when the pressure is under critical and the described safeguard depends on the pressure to be critical to be evaluated.

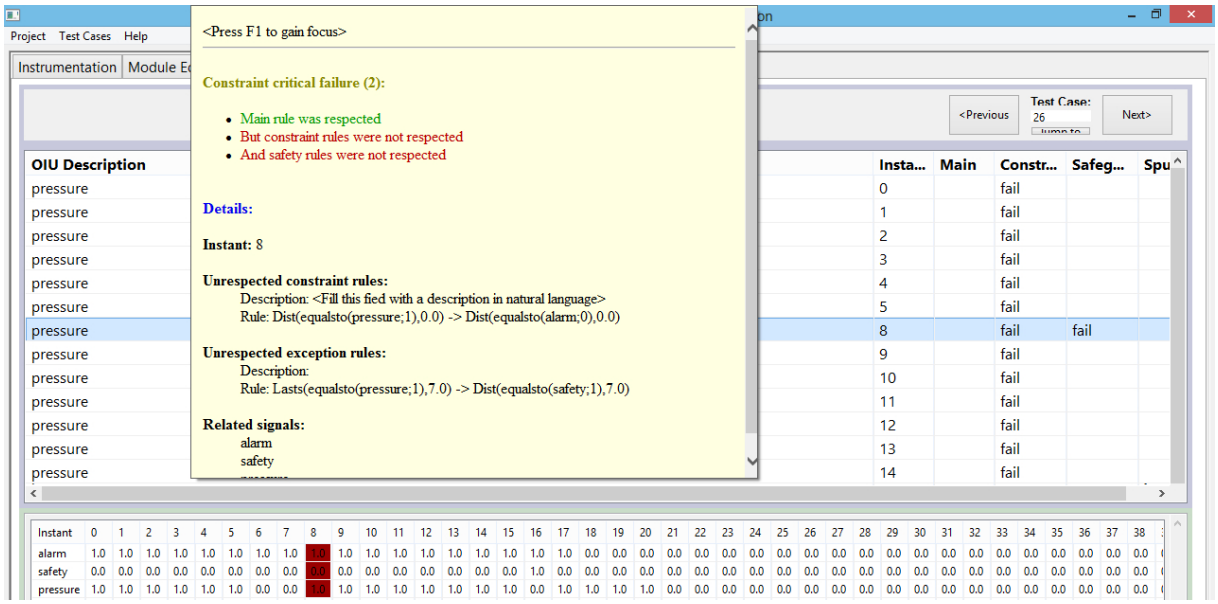


Figure 5.22: Oracle report: constraint critical failure

But, forcing such combination of violation by changing one of the rules, a constraint critical violation can be detected, as shown in Figure 5.22.

5.5 Final Remarks

This chapter presented an overview of Apolom, an oracle generator tool that provides support to activities of the oracle definition process presented in Chapter 4. The manual and automated features supported by the tool were described and a running example was provided to a better understanding about its functionalities.

The main contribution of our tool is twofold: it evidences the feasibility of an oracle generator implementation, respecting the previously proposed process; and, it is a vehicle to the execution of four studies (Chapter 7) which assess the viability of our approach.

Next chapter details the implementation and limitation of Apolom.

Chapter

6

Apolom: Implementation and Limitations

Apolom was implemented by the author of this thesis, in Java, for the period of one year. It contains approximately 25000 lines of code and 525 classes which are grouped into three main packages:

- bridge: implements the mapping between model and oracle;
- oracleinformation: implements the structure of the specification as OIU, Behavior, Expression classes, and specification editors;
- oracleprocedure: implements the analysis engine, language parser and data access.

Next three sections describe each package in details. Section 6.4 presents Apolom limitations.

6.1 The *bridge* Package

Simulink inspired the development of similar tools, as Scicos and XCos, both free software. The implementation of Apolom was intended to facilitate the support to such tools in the future. In this way, an intermediary model was defined, called ApolomTree, which

represents a generalization of Simulink, Scicos and XCos models. The drawing of a model in Apolom, its navigation and management of the model instrumentation are set in the intermediary model.

ApolomTree is a structure with access methods in which nodes represent blocks of the original model and edges correspond to lines. Both elements contain basic properties to allow the drawing of a model in Apolom and a unique identification of each block and line for the instrumentation.

The bridge is composed by three packages:

- *Data*: represents ApolomTree model and a mapping table;
- *Driver*: converts a Simulink model into ApolomTree;
- *Connector*: maps the relation between ApolomTree and the oracle specification attributes. It is also responsible by the instrumentation of the models.

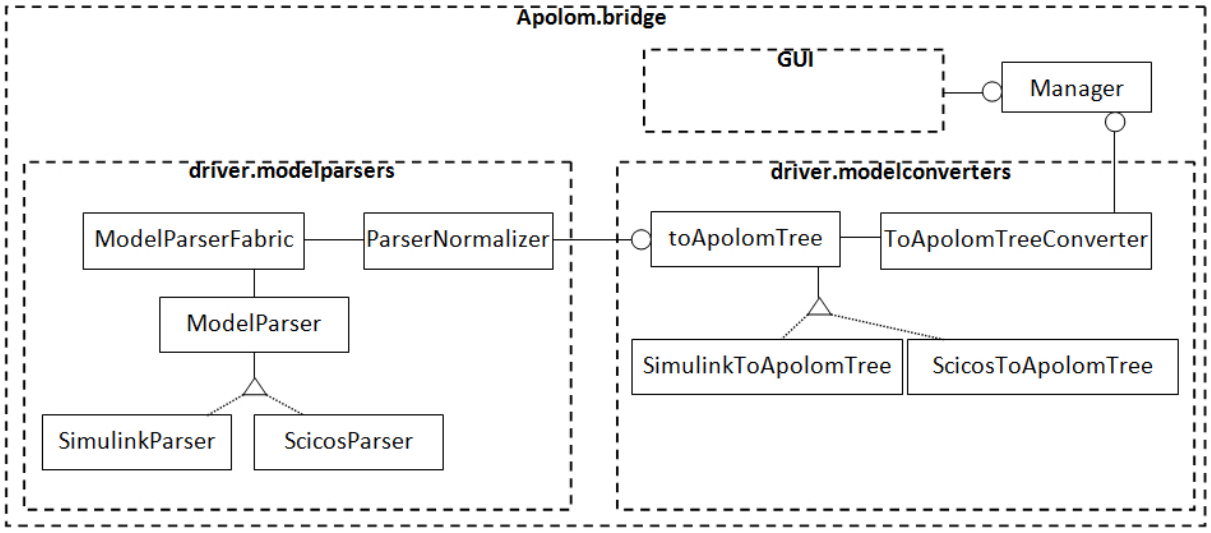


Figure 6.1: Driver package: parsers and converters

Figure 6.1 presents a simplification of *driver* package. *Modelparsers* implements parsers for each original format (Simulink and Scicos in the current version), and an interface with a converter. *Modelconverters* is the package responsible by the transformation of an original model to ApolomTree.

Connector implements the instrumentation and mapping between model and specification. The pattern of the instrumentation is similar to the one presented in *driver* package, that is, a factory instantiates an instrumenter object based on the model format. However, only Simulink is supported in the current version.

6.2 The *oracleinformation* Package

This package implements the layers of the oracle information – modules, OIUs and behaviors – and their respective editors and Rule Wizard.

The modules are implemented as a tree structure in which a node is a *Module* class (Figure 6.2). Each node has a hashset of module children, a father, name and a code.

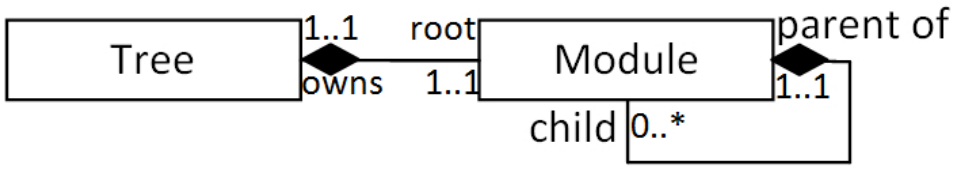


Figure 6.2: Module structure

A behavior is composed by its name, description, signal names and values, parameter names and values, an analysis array and an expression. When its expression is analyzed, the oracle stores, in the array, whether the behavior holds for each instant.

An OIU component represents a main rule, constraint or safeguard. It is composed by an expression (which represents a rule in TRIO/Apolom), a reference to the set of behaviors present in the expression and an array which represents whether the expression holds for each instant. An OIU is composed by a description, a main OIU component, and two sets of OIU components representing the respective constraints and safeguards. An OIU description contains a mapping between expressions and their descriptions for the main rule, as for all constraints and safeguards. It also contains an unique cod and a reference to its module. The relation between such classes is illustrated in Figure 6.3.

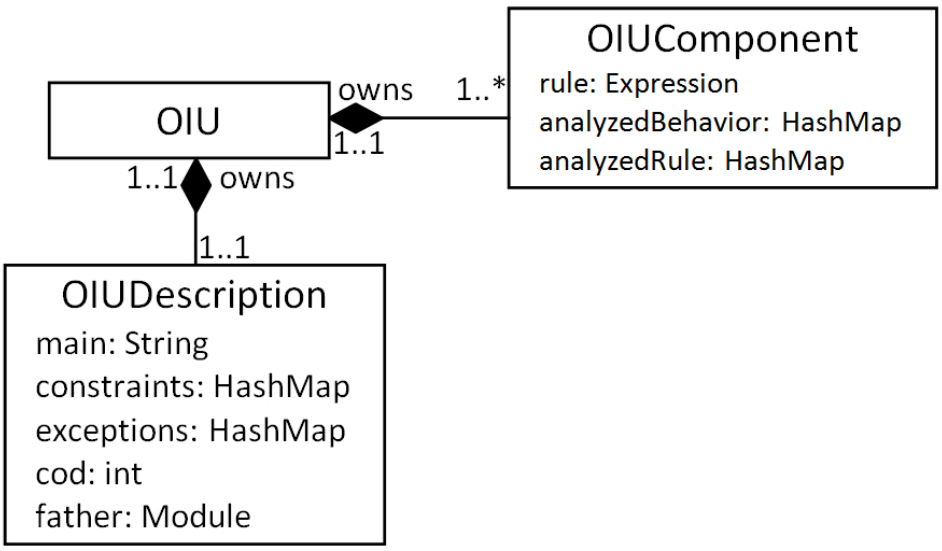


Figure 6.3: OIU structure

The OIU class is responsible only for the OIU result calculation, that is, if such unit passed in the test or, otherwise, the type of violation (Section 4.2.2, page 48). The OIU description encompasses the tracing between each rule and its natural language description. These relations could be implemented directly in the OIUComponent, but they were decoupled from such class as a design decision: it is intended to expand the tracing from the rules to documents outside the oracle generator, as future work, and the decoupling may avoid changing essential code.

6.3 The *oracleprocedure* Package

This package implements the analysis engine, the parser of the specification and the data access. The engine contains a main controller, behavior analyzer, OIU analyzer and a support analyzer.

Main controller defines the engine control flow and the execution of each analysis, as follows:

```
1.  input: OIU list;
2.  extract all behavior-sequence pairs from each OIU in the list;
3.  for each instant                          //behavior analysis
4.      execute the analysis of each extracted behavior-sequence pair
5.      output: behavior analysis
6.  for each instant                          //OIU analysis
7.      for each OIU from the list
8.          execute the analysis of all rules;
9.          output: violation report for the current OIU analysis;
10. if analyze_spurious
11.     for each instant   //Support analysis
12.        retrieve the next OIU from the list;
13.        retrieve all behavior-sequences from the current OIU;
14.        for each behavior-sequence
15.       if it holds at the current instant
16.              verify if it is spurious
17.              output: spurious analysis at the current instant
```

Each analysis (behaviors, rules and spurious values) is performed in sequence. When analyzing a rule, all behaviors were already analyzed. Therefore, if more than one rule references a same behavior, it will not require multiple analysis of the same behavior.

The **OIU analysis** is presented in more details in the next code:

```
1.    analyze the main rule of the current OIU;
2.        if it is the right side of an expression
3.            mark the referenced behavior-sequence pairs;
4.    analyze the set of constraints of the current OIU;
5.        if it is the right side of an expression
6.            mark the referenced behavior-sequence pairs;
7.    analyze the set of safaguards;
8.        if it is the right side of an expression
9.            mark the referenced behavior-sequence pairs;
10.   if success, return 4;
11.   if only constraint fail, return 3;
12.   if only constraint and safeguard fail, return 2;
13.   if only main fail, return 1;
14.   if main fail and safeguard fail, return 0;
```

An expression may have an *implies* connector. In such case, spurious occurrences can be identified if behavior-sequence pairs of the right side of the connector are not related with the left side. The detection of spurious occurrences is described in details in the next section.

6.3.1 Detection of Spurious Occurrences

To describe the detection of spurious occurrences (Section 4.4.3, page 57), a concrete example and the mechanism of detection is presented.

Concrete example

To this example, the requirement of Section 4.4.3 and two examples of behavior-sequence pairs are used:

(3) $Greater(sequence; value) = sequence > value$

(4) $Descendant(sequence) = \neg AtEnd(sequence) \wedge Current(sequence) > Next(sequence, 1)$

Be A an instance of pair model (3):

Greater(seq_a;10)

In the example, A is a pair composed by a behavior *greater than 10* and sequence *seq_a*. B is an instance of pair model (4), where a sequence *seq_b* has descendant value:

Descendant(seq_b)

The requirement of Section 4.4.3 can be expressed as the following TRIO formula:

$$\textbf{(5)}\ Starts(A) \wedge howlong = NowOn(A) > 0 \implies \exists(Lasts(B, howlong - 1), 0, 5)$$

The left side of the formula expresses that in the instant when A starts to hold, the variable *howlong* will represent how much instants A holds in sequence. It means that the oracle procedure will analyze in which instants A will hold for some interval and for how long. When the oracle finds such scenario, it will analyze the right side of the formula which expresses that exists in the next 5 instants, a time where B lasts for the same interval of A.

Mechanism of spurious occurrence detection

As described before, the oracle procedure will analyze such scenario differently, depending on the adopted assumption. For this example, let us consider *successive* assumption, where the oracle must report that instant 24 has a spurious occurrence of B.

The spurious detection is executed in two steps: (i) during the OIU analysis, the procedure marks in which instants a pair behavior-sequence of the right side of an *implies* connector is used in the evaluation of a rule; and, (ii) in the spurious analysis, the procedure reports in which instants this same pair holds but it is not marked as used in the rule evaluation. To illustrate the presented mechanism, let us consider the concrete example at each oracle analysis.

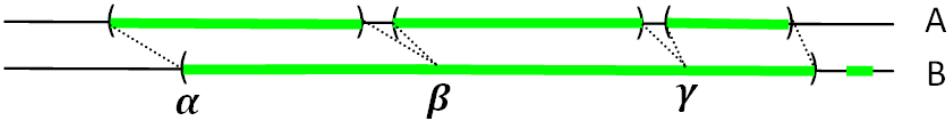


Figure 6.4: Successive assumption analysis

The **behavior analysis** generates an array for each A and B pairs, with the instants that they hold (green in Figure 6.4).

During the **OIU analysis**, the oracle procedure uses the results from the behavior analysis, relates the occurrences of B with the occurrences of A and it does not allow that any occurrence of B be related with more than one occurrence of A.

For example, in Figure 6.4, α is the instant when the first occurrence of B is related with the first occurrence of A and β is the last occurrence of B which is related with the last instant of the first interval of occurrences of A. Because the oracle assumption is set as *successive*, the first occurrence of the second interval of occurrences of A will be

related with the next instant when B holds, after instant β. The last occurrence of B is not related with any occurrence of A, thus, it is considered as a spurious value.

More specifically, based on the concrete example: the oracle procedure uses the results from the behavior analysis to evaluate rule (5): it analyzes if pair A starts to hold at the current instant, that is, A must hold at the present analysis instant but must not held at the immediatly past instant. Also, operator *NowOn()* returns to variable *howlong* how much instants A holds in sequence, when it starts to hold. For example, in Figure 4.7, at analysis instant 0, the left side of rule (5) is true and operator *NowOn()* returns 7. The left side is also true at instants 9 and 18, with *NowOn()* returning respectively 8 and 4.

If A starts to hold at a given analysis instant, the procedure analyzes the right side whether B holds for the same number of instants of A, in the next 5 instants. Considering Figure 4.7, at instant 0, the right side is true because there is a time in the the next 5 instants (at instant 4 in the future) where B holds for 7 instants (from instant 4 to 10). In such case, the procedure registers in an array that B was analyzed in the right side for such rule at instants 4 to 10.

The same occurs at instant 9 and 18, when the left side is true again. But because successive assumption is in use in this example, the procedure will ignore the instants in which B was already marked as analyzed. For example, at instant 9 the left side is true (A holds for 8 instants) then the right side must be analyzed. At this point, the procedure will search if B holds for 8 instants within the next 5 instants, starting from instant 9 – the current analysis instant. However, because B was already analyzed from instant 4 to 10, the procedure will ignore instants 9 and 10 (when B actually holds, but is marked as analyzed) and starts to analyze the right side from instant 11. Finally, because B holds from instant 11 to 18, the right side is true, such instants are registered in the array B as analyzed and the rule is respected. The same concept is applied at instant 18.

In the end of the OIU analysis, an array, bound to B and the current rule marks all instants when B was used in the analysis. In the example, the array registers instants 4 to 22: interval from instant 4 to 10 of B is registered at analysis instant 0, when the left side is true for the first time in the OIU analysis; interval from 11 to 18 of B is registered when the left side is true at instant 9; and interval from 19 to 22 of B is registered at analysis instant 18. One must note that instant 24 of B is not registered as used.

The **spurious analysis** compares such an array with the result of the behavior analysis of B. It analyzes if there is an instant when B holds (behavior analysis) but is not registered as used in the array created in the OIU analysis. If so, a spurious occurrence is detected.

6.4 Limitations

Apolom is a tool designed and implemented by one person for the period of one year. Its limitations are mostly regarded to time restriction and programming skills.

Interface limitations: the Rule Wizard does not allow the selection of more than one attributes for the same behavior, if it is required. Therefore, the tester must add the remaining attributes to the generated expression. For example, behavior *compares* requires two sequences:

compares(seq1,seq2;precision)

But, the Wizard mechanics allows the selection of only one attribute. It can be overcome, in the future, by allowing the selection of multiples items within the list of attributes.

Also, the tool was designed with SWT, an SO dependent widget toolkit. It implies that the interface layout may differ from a SO to another.

A third interface limitation concerns to the presentation of the Simulink model. Although it represents the block size and position accurately, the block picture may not reflect the correct layout, drawing it as a generic white rectangle with black border. The lines, also, may not be drawn with precision. Because the tool was designed to support other models in the future, as Scicos, the model reading was standardized. A built-in parser was designed by re-engineering and it is possible that unanticipated elements from the model cause a fail in its reproduction.

Finally, it is not possible, for now, to edit the module or OIU position in the module tree, after creating it.

Model changing: if a model is changed over the testing activity, it should not cause consequences to the oracle, except if a mapped line is removed. To avoid instrumentation errors, the tester must unmap the line from Apolom before removing the line in the original model.

Language implementation: TRIO/Apolom was implemented with JavaCC. This parser generator version has a limited backtracking support. An ambiguity was not solved in time in relation to the use of parenthesis surrounding logical expressions and basic operations. The limitation was solved by using brackets around logical expressions instead of parenthesis. The language also does not allow recursion or behaviors within behaviors.

File format incompatibility: Apolom supports Matlab 7.7.0 (version R2008b). New Matlab versions use different file formats to log signal values which are not compatible

with Apolom. To bypass this issue using more recent Matlab releases, the generated files must be loaded into Matlab and saved as an older file format.

Step solver: a solver is a Simulink software which determines the time of the next simulation step. It has two modes: fixed-step and variable-step. The designer must select fixed-step if the model is planned to be deployed, because one can not map the variable-step size to the real-time clock. If the model is not intended to be deployed, variable-step solver may be selected to shorten the simulation time in exchange for less accuracy. Apolom only supports fixed-step solver.

We consider that all described limitations can be overcome and do not threaten the approach feasibility.

6.5 Final Remarks

This chapter presented the implementation and limitation aspects of Apolom, an oracle generator tool which represents an instance of this research. Its structure is composed by three main packages: *bridge*, *oracleinformation* and *oracleprocedure*, responsables by the mapping, oracle information representation and analysis, respectively.

Software testing is usually not practical or effective without support tools. The main contribution of this chapter is to present the feasibility of developing a tool that supports all proposed oracle definition activities. It also discusses key points of the implementation, which may guide the development of similar tools or future extensions, as support for new specification languages.

The discussed limitations of Apolom can be explained by the development conditions: time and resources. Although the limitations indicate that this tool should be improved to increase its effectiveness and usability, it also may evidence that a release tool could be developed with no much effort by a small team within a short period with low cost.

Next chapter employs Apolom to empirically evaluate the proposed solution.

Chapter

7

Empirical Evaluation

This chapter presents four evaluation studies to verify the soundness of the proposed solution. It aims to analyze (i) whether an oracle generator tool is feasible; (ii) whether oracle specification is practical; (iii) whether instrumentation added by the mapping interferes with the actual simulation and delays it, that is, the probe-effect (McDowell and Helmbold, 1989); (iv) whether analysis time and resources are acceptable; and, (v) whether an oracle for Simulink models is effective.

7.1 Issues and Hypothesis Statements

The objective of this section is providing a clear comprehension of each analyzed concern. The issues, raised during the research, are here structured as presented in Table 7.1.

Table 7.1: Issues and hypothesis statements

Issue 1	There is no guarantee that the approach can be partially automated as it is stated.
Question	Is an oracle generator tool actually deployable?

Hypothesis statements	• H_0: an oracle generator tool is not deployable. • H_1: an oracle generator tool is deployable.

Issue 2	The simulated model that will be analyzed by the oracle is an instrumented copy of the original. Instrumentation may cause probe-effect.
Question	Can instrumentation affect the simulation time significantly?
Hypothesis statements	• H_0: the instrumentation affects the simulation time significantly. • H_1: the instrumentation does not affect the simulation time significantly.

Issue 3	A Simulink-like model is a high-level representation of a system and a formal specification language may not be attractive to be used.
Question	Can a specification language be simple enough to be adopted by the industry? • Is a specification language always complex in comparison with a Simulink-like model? • Is it always hard to elaborate?
Hypothesis statements	• H_{0a}: the oracle specification language is always complex in comparison with a Simulink-like model. • H_{1a}: the oracle specification language may be simpler than a Simulink-like model. • H_{0b}: the oracle specification language is hard to elaborate. • H_{1b}: the oracle specification language may be simple to elaborate.

Issue 4	The oracle may take too much time and resources to analyze the simulation against the specification.
Question	Is the oracle analysis time and resources prohibitive?

Hypothesis statements	• H_0: The oracle analysis time and resources are, respectively, too long and prohibitive. • H_1: The oracle analysis time and resources are acceptable.
Issue 5	An oracle generation demands effort from a testing team. At least one tester must learn an oracle specification language and how to operate a tool as Apolom. After becoming a specialist, he/she must spend time to write a specification. An oracle generation may not be attractive if its analysis effectiveness is not better than manual comparison.
Question	Can generated oracles be effective?
Hypothesis statements	• H_0: The oracle analysis is not better than manual analysis. • H_1: The oracle analysis may identify failures which would not be easily found manually.

To answer the questions from Table 7.1, Apolom was implemented (Chapter 5), and four evaluation studies were executed and detailed in the next sections. The implementation of Apolom supports hypothesis H_1 from question 1.

7.2 Evaluation 1

This study aims to answer question 2 and 3. The independent variables are: Apolom tool, the system description, student expertise, training time and hardware. The dependent variables are: the model, the time to develop the oracle specification, time to find errors, number of errors, oracle analysis time, execution of the original model and execution of the instrumented model.

7.2.1 Operation

Sixteen PhD students participated on this experiment, in which 8 were trained in Simulink and Apolom and 8 had no knowledge on both participating only in the study of the language acceptance, Section 7.2.3. The trained group was asked to develop a Lorenz Attractor on Simulink, based on the following system description: “the Lorenz Attractor is composed by three equations”:

$$x_{i+1} = x_i + (-10 * (x_i - y_i)) * 0.001 \tag{7.1}$$

$$y_{i+1} = y_i + (-x_i * z_i + 28 * x_i - y_i) * 0.001 \tag{7.2}$$

$$z_{i+1} = z_i + (x_i * y_i - 8 * z_i/3) * 0.001 \tag{7.3}$$

Two students did not model the specification correctly with Simulink.

The oracle specification was written by the author of this thesis – which plays the role of the tool specialist. As example, Figure 7.1 represents a Simulink model of Lorenz equation (2).

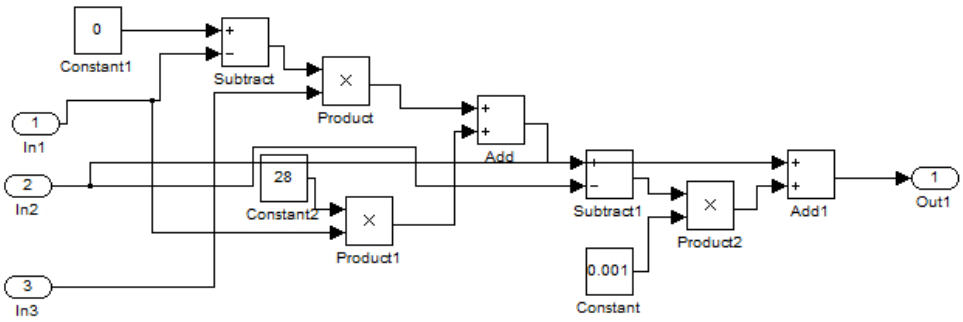


Figure 7.1: Lorenz Equation

The specification attributes were defined and mapped (Figure 7.2). The tester selected the signals directly from the model visualization and mapped them to the respective *specification attributes* list by selecting them and pressing the *insert* button.

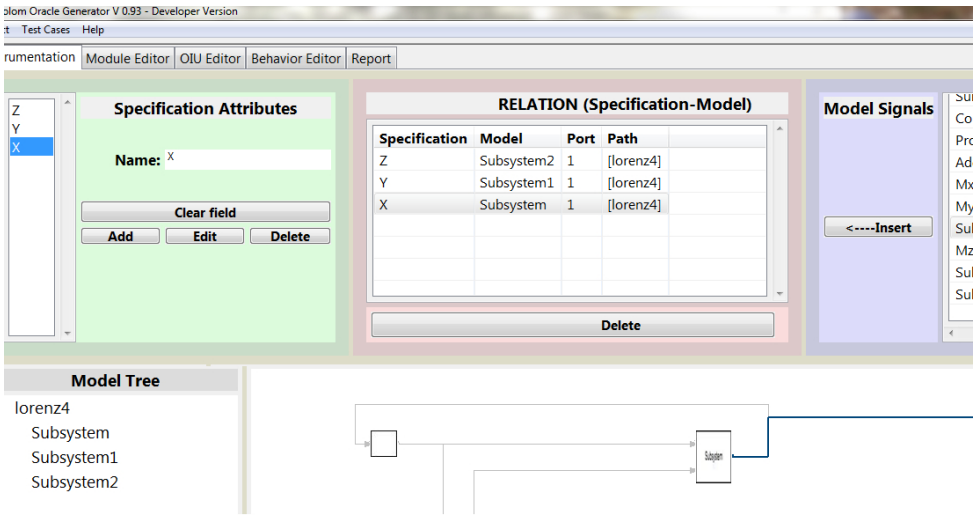


Figure 7.2: Attribute definition and mapping

Three behaviors were defined, one for each Lorenz equation. Figure 7.3 presents the representation of equation (2), named as LorenzY.

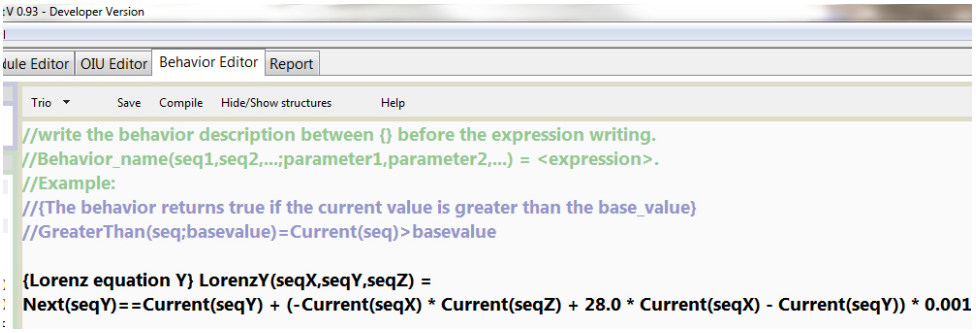


Figure 7.3: Behavior of equation (2)

After both models and oracle information are ready, the instrumented models were executed. Next subsections details each evaluation.

7.2.2 Simulation Time of Instrumented Models

To explore the influence of the instrumentation in the simulation time and answer open question 2, thirty-two simulations were executed on the original and the same number on the instrumented model. Each simulation was configured with a total of 180,000 steps. In a discrete real-time system, in Simulink, 1 second represents 5 steps by default – one step represents 0.20 seconds. Then, 180,000 steps represent 1 hour of real-time simulation. The hypotheses statements were translated to:

- H_0: there is a difference between the average simulation time from the original model and from the instrumented model ($\mu_{original} \neq \mu_{instrumented}$).
- H_1: there is no real difference between the average simulation time from the original model and from the instrumented model ($\mu_{original} = \mu_{instrumented}$).

There are three instrumented signals for the given specification, one for each variable (x, y and z). The simulation time means and standard deviations for both original and instrumented models were $\mu_{original} = 14.5797$, $\sigma_{original} = 0.0687$, $\mu_{instrumented} = 14.6047$ and $\sigma_{instrumented} = 0.0779$. A two-tail test, with $\alpha = 0.05$, showed that both samples are statistically equivalents with ρ-value $= 0.1782$. However, a threat to validity must be eliminated: the three instrumented signals may not be significant to generalize the experiment results. To allow a better generalization, one more case was created to increase the number of instrumented signals to 20, representing 51.28% of the total signals in the model. The instrumented copies were simulated 32 times, each, with means and standard deviations of $\mu_{original} = 14.53$, $\sigma_{original} = 0.0967$, $\mu_{instrumented} = 14.5312$ and $\sigma_{instrumented} = 0.0638$. After the executions, the two-tail test was applied to the new results. It did not show statistical difference in relation to the original model, with $\alpha = 0.05$ and ρ-value $= 0.9515$.

Both studies support hypothesis H_1 from open question 2.

7.2.3 Language Acceptance

All students were submitted to the following question: "considering the original equation (2) and its respective representation of Figure 7.1 and Figure 7.3, which one is easier to read and write?" Seven of eight trained students answered the LorenzY representation, which supports the hypothesis H_{1a} and H_{1b} from open question 3. The same question was presented to the second group and all students answered the LorenzY representation as the easier way to read and write the original requirement.

7.2.4 Threats to Validity

A **threat to external validity** is a condition that limits the ability to generalize the results of an experiment to industrial practice (Wohlin et al., 2000).

There is a possible threat to external validity on issue 3 (H_a) analysis. An equation was translated to a Simulink model and an oracle specification language, and compared both readability. It may not be always easier to write systems specifications with the oracle language if compared to a Simulink representation. In fact, the objective of the experiment is not proving such statement, but demonstrating that it can be easier. And, with a tool support, as the wizard presented in Figure 5.1 (page 60), it can be soften.

Another threat to external validity is the influence of the instrumentation over the simulation time on more complex models. The analysis demonstrated that there is no relevant variation between the original and instrumented model, when 20 signals are instrumented in a small model. More complex models with more instrumented signals may be used to allow a better generalization of the results.

7.3 Evaluation 2

The second study focused on questions 3 (mostly H_b) and 4. An introduction to the model and the simulation is presented, as the formal and informal requirements and the common development team concerns about the usual validation of the model. The oracle specification is also presented. The results are based on the following dependent variables: time to develop the oracle specification, analysis time, overall time and the number of detected simulation errors.

7.3.1 Simulation

An UAV (Unmanned aerial vehicle) model was granted by AGX Technologies [1], under a confidentiality agreement, as part of Tiriba Project (Branco et al., 2011). It is a large-scale model (Mathworks, 2011) (Shailesh, 2011), with 6111 blocks. An aviation glossary is available in Appendix A.

The vehicles are commercially known as AGPlane. These aircrafts are extensively used in agricultural and environmental monitoring, but they can also be applied for security and civil defense.

The model simulates an autonomous flight based on a preset flight plan consisted by four waypoints. The flight plan is hard-coded in a block that represents the UAV memory.

The UAV must reach all waypoints, in sequence from 1 to 4 and return to 1. This route must be accomplished twice. If the current course is in a closed-angle w.r.t. the next waypoint, the UAV executes a correction maneuver (also called loop).

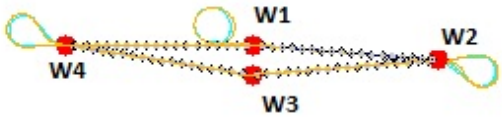


Figure 7.4: UAV simulation: waypoints

Figure 7.4 is presented by Simulink during the simulation. The red circles are the waypoints. The green line is the expected route and the yellow line is the actual plane route. The UAV starts close to waypoint 1 (*W1*) and executes a correction maneuver to pass through *W1* in a straight line in relation to the next waypoint (*W2*). When it reaches *W2*, it also executes a correction maneuver to align itself in relation to waypoint 3 (*W3*). When it reaches *W3* it does not need to correct itself with a loop because the course w.r.t. the next waypoint has an obtuse angle. After *W4*, it returns to *W1* and repeats the same paths one more time.

7.3.2 System Description

The specification reflects both formal and informal requirements used by the model development team to evaluate model consistency and simulation. It was divided in three groups: safety requirements, waypoint requirements, and model consistency requirements.

Figure 7.5 presents a simplified representation of the model subsystems to help the visualization of the next requirements (most names were eliminated to respect the confidentiality agreement).

[1] http://www.agx.com.br

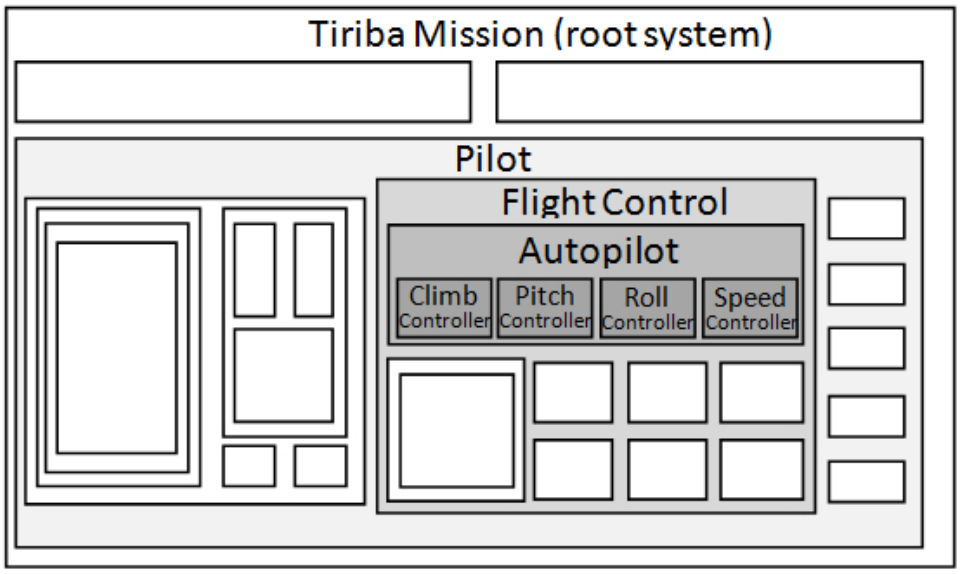


Figure 7.5: UAV simplified subsystem diagram

Safety requirements (R1)

1. The roll angle must be limited to 40 degrees;
2. The angle of attack may never exceed 20 degrees;
3. The airspeed must always be greater than 1.05 the stall velocity (Bennani and Looye, 1998). $V_{stall} = 16.67m/s$ and $V_{safe} = 17.50m/s$;
4. The vertical acceleration should be limited to $10m/s^2$.

Waypoint requirements (R2)

1. Each waypoint is represented by tree coordinates: longitude (degrees), latitude (degrees) and altitude (meters). The flight plan is composed by four waypoints:
   - Waypoint 1 (*W1*): -122.3700, 37.6300 and 100;
   - Waypoint 2 (*W2*): -122.3690, 37.6400 and 100;
   - Waypoint 3 (*W3*): -122.3680, 37.6300 and 100;
   - Waypoint 4 (*W4*): -122.3700, 37.6200 and 100.
2. The plane must pass through each waypoint, in sequence, and return to *W1*;
3. The same route must be executed 2 times;
4. Because the same route must be executed twice, *W3* will be reached 2 times;

5. Because the same route must be executed twice, *W1* will be reached 3 times;
6. Considering that a correction maneuver will make the plane pass through the same waypoint twice, it is expected that *W2* and *W4* will be reached for 2 times each for a completed circuite. Therefore, these waypoints will be visited 4 times each in the simulation;
7. The flight time from *W1* to *W2* and from *W3* to *W4* must be less than 50 seconds;
8. The flight time from *W2* to *W3* and from *W4* to *W1* must be less than 90 seconds.

Model consistency requirements (R3)

1. The roll value must be the same in the *Pilot* subsystem and in the *Roll Controller* subsystem;
2. The lever smoothing roll value must be the same in the *Pilot* subsystem and in the *Roll Controller* subsystem, with a tolerance of 2 degrees;
3. The pitch value must be the same in the *Pilot* subsystem and in the *Pitch Controller* subsystem;
4. The airspeed value must be the same in the *root* system and in the *Speed Controller* subsystem, with a tolerance of $0.2m/s$;
5. The climb value must be the same in the *Pilot* subsystem and in the *Climb Controller* subsystem;
6. The difference between the autopilot roll command and the actual roll should not be greater than 1 degree. It will be considered critical if it stands greater than 1 degree for more than 5 seconds or if it is greater than 2 degrees;
7. The difference between the autopilot pitch command and the actual pitch must be less than 1.5;
8. The difference between the autopilot airspeed command and the actual airspeed must be less than $5m/s$;
9. The difference between the autopilot climb command and the actual climb must be less than 1 degree.

7.3.3 Common Development Team Concerns

In interview, three drawbacks were emphasized by the development team: (i) "It is not convenient to zoom in and zoom out the flight route to validate the simulation"; (ii) "one has to follow the simulation to verify if the course is right and if the plane behaves consistently with the expected"; (iii) "Sometimes an adjustment is made in the model and it is interesting to observe how it reflects in the simulation. We identify the signals of interest and we use a scope block or a graphical block to make the observation. But it is not practical to check manually".

To evaluate the distance in which the UAV pass through a waypoint, the tester may zoom in (i) the image drawn in the simulation. This must be repeated at all waypoints and, yet, a precise distance between UAV and waypoint is hardly achieved by visual observation. Also, the UAV must repeat the same route twice and the line which represents the UAV flight may be overlapped, making it difficult to visualize.

If the tester does not follow the simulation (ii), it is hard to evaluate whether the UAV passes from a waypoint to other within the expected time and whether it flights as expected.

The evaluation of the UAV simulation contains the following features:

- Its execution represents 10 minutes of flight;
- The simulation may take more than 17 minutes to be executed, depending on the hardware resources;
- Considering only the signals associated with the system requirements presented in the last section, the simulation produces over 741,031 output data to be evaluated.

The number of requirements is substantially large to be evaluated manually, considering the amount of generated outputs. Temporal requirements hamper the test activity because the tester may need to check values in different points of the simulation. Also, the simulation time may not be in proportion of 1:1, that is, 10 minutes of simulated UAV flight may take 17 minutes to be accomplished. It makes difficult to check temporal properties by observation.

All these discussed drawbacks make manual evaluation an ineffective activity and favor the use of partially-automated oracle generators.

7.3.4 Oracle Specification

To translate the system description to the oracle specification (which corresponds to the information definition from the proposed process), four steps were performed: the oracle

specification attribute identification, the modularization, the OIU definition and the behavior definition.

Oracle specification attribute and modularization

As the first step, the oracle specification attributes were identified, based on the system description:

- In $R1\_1$: roll angle from the root subsystem;
- In $R1\_2$: angle of attack from the root subsystem;
- In $R1\_3$: airspeed from the root subsystem;
- In $R1\_4$: vertical acceleration from the root subsystem;
- In $R2$: longitude, latitude and altitude (waypoint coordinates);
- In $R3\_1$: pilot subsystem roll (roll angle from $R1\_1$) and controller roll;
- In $R3\_2$: pilot subsystem lever smoothing roll command and controller lever smoothing roll;
- In $R3\_3$: pilot subsystem pitch and controller pitch;
- In $R3\_4$: pilot subsystem airspeed and controller airspeed;
- In $R3\_5$: pilot subsystem climb and controller pitch;
- In $R3\_6$: autopilot roll command and actual roll (roll from root subsystem);
- In $R3\_7$: autopilot pitch command and actual pitch (pitch from root subsystem);
- In $R3\_8$: autopilot airspeed command and actual airspeed (airspeed from root subsystem);
- In $R3\_9$: autopilot climb command and actual climb (climb from root subsystem).

The second step is the module creation, one for each requirement group: safety, waypoints and model consistency.

Behavior and OIU definition

The third step was the OIU creation. Each OIU was created within its respective module. All Trio expressions were generated with Rule Wizard, except for $R2\_4$, $R2\_5$,

$R2\_6$ and $R2\_8$ because the concept of *X times in the simulation* was not previously foreseen. However, such concept can be inserted as a new Rule Wizard *shortcut*.

$$R1\_1 : Dist(lessequalsthan(roll; 40), 0)AND$$
$$Dist(greaterequalsthan(roll; -40), 0)$$

In the expression $R1\_1$, $Dist(\langle behavior \rangle, 0)$ means that a behavior must be true at the analysis instant. Therefore, the roll angle must be less than or equals to 40 and greater than or equals to -40 at the analysis instant.

$$R1\_2 : Dist(lessequalsthan(attack; 20), 0)$$

The expression $R1\_2$ means that, at the analysis instant, the angle of attack must be less than or equals to 20.

$$R1\_3 : Dist(greaterthan(airspeed; 17.50), 0)$$

The expression $R1\_3$ means that, at the analysis instant, the airspeed must be greater than $17.50m/s$.

$$R1\_4 : Dist(greaterequalsthan(verticalacc; -10), 0)$$
$$ANDDist(lessequalsthan(verticalacc; 10), 0)$$

The expression $R1\_4$ means that, at the analysis instant, the vertical acceleration must be greater or equals to $-10m/s^2$ and less than or equals to $10m/s^2$.

$$R2\_1, R2\_2 and R2\_7 : Starts($$
$$geocoord(longitude, latitude, altitude;$$
$$-122.37, 37.63, 100, 0.0001, 0.0001, 10)) - >$$
$$Exists(Dist(geocoord(longitude, latitude, altitude;$$
$$-122.3690, 37.64, 100, 0.0001, 0.0001, 10), 0), 0, 2500)$$

Previous expression partially represents requirements $R2\_1$, $R2\_2$ and $R2\_7$. It means that, if the UAV pass through waypoint 1, there is an instant in the next 2500 instants (50 seconds) when it will pass through waypoint 2.

The behavior *geocoord* returns true if the actual UAV's longitude, latitude and altitude are equals to the given values, within a margin of error of 10 meters for each coordinate. The first three parameters correspond to the oracle specification attributes, i.e., the current UAV position. Next three parameters correspond to the coordinates to be reached by

the UAV (a waypoint). The last three parameters correspond to the margin of acceptable error (10 meters or 0.0001 degrees). In the simulation, a second is divided in 50 instants. In this case, 50 seconds means 2500 instants. The behavior is expressed as:

$$\begin{aligned} B1 : geocoord(longitude, latitude, altitude; \\ xcoord, ycoord, zcoord, xprecision, yprecision, zprecision) = \\ [Current(longitude) <= xcoord + xprecision AND \\ Current(longitude) >= xcoord - xprecision AND \\ Current(latitude) <= ycoord + yprecision AND \\ Current(latitude) >= ycoord - yprecision AND \\ Current(altitude) <= zcoord + zprecision AND \\ Current(altitude) >= zcoord - zprecision] \end{aligned}$$

With the same concept, three more OIUs were created to express the flight between waypoints 2 to 3, waypoints 3 to 4 and waypoints 4 to 1.

$$\begin{aligned} R2\_4 : Clock(longitude) == 0 -> ExistsQ(Starts(\\ geocoord(longitude, latitude, altitude; \\ -122.3680, 37.6300, 100, 0.0001, 0.0001, 10) \\ ), 0, 30000) == 2 \end{aligned}$$

$ExistsQ$ returns the number of times that an expression holds in an interval of time. In the above expression, this operator returns the number of times that the UAV passes through a waypoint from the analysis instant to the end of the simulation. $Clock$ returns the simulation clock at the analysis instant. It guarantees that the right side of the expression will be analyzed just once, in the first analysis instant. Otherwise, the whole expression would be analyzed for all the simulation instants, which is not required.

The remaining waypoint requirements ($R2\_5$ and $R2\_6$ and $R2\_8$) are similar to $R2\_4$.

$$R3\_1 : Dist(compares(roll, roll\_controller), 0)$$

Requirements from $R3$ group check the model consistency. Expression $R3\_1$ compares the actual UAV roll in two points in the model: in the pilot subsystem and in the roll controller subsystem. They are supposed to have the same values. The same concept is

applied to requirements $R3\_2$, $R3\_3$, $R3\_4$ and $R3\_5$. The *compares* behavior is built-in on Apolom.

$$R3\_6 : Dist(greaterequalsthan(rolldifference; -1), 0)$$
$$ANDDist(lessequalsthan(rolldifference; 1), 0)$$

$R3\_6$ indicates that the roll difference between the autopilot roll command and the actual roll command should be less than or equals to 1 and greater than or equals to -1. there is, also, two other requirements that are closely related to R_6: (i) if such difference is greater than 1 for more than 5 seconds or (ii) if the difference is greater than 2 degrees, it is considered a critical failure.

$$R3\_6_{s1} : NowOn(lessthan(rolldifference; -1)) < 250$$
$$ANDNowOn(greaterthan(rolldifference; 1)) < 250$$

Expression $R3\_6_{s1}$ was defined as a safeguard, and it represents requirement (i).

$$R3\_6_{s2} : Dist(greaterequalsthan(rolldifference; -2), 0)$$
$$ANDDist(lessequalsthan(rolldifference; 2), 0)$$

The safeguard expressed by $R3\_6_{s2}$ represents requirement (ii).

The same construction of $R3\_6$, with different attributes and values, can be applied to requirements $R3\_7$, $R3\_8$ and $R3\_9$.

7.3.5 Results

The evaluation has two goals: investigating (i) whether the oracle specification elaboration is practical or not, and (ii) whether the time and resources spent in the oracle analysis are acceptable or not.

The oracle specification was created by the PhD student to analyze an industrial, large-scale, model. The time to develop it, including the textual explanation of each rule, was 54 minutes and 45 seconds. The partial times were: 2 minutes and 37 seconds to define the attributes, 8 minutes and 41 seconds to map the attributes to the model, and 43 minutes and 27 seconds to define the OIUs and their descriptions. The average time to translate a system requirement as part of the oracle specification was approximately 2

minutes and 30 seconds. Although a reference could not be found, it seems an acceptable analysis time.

The elapsed time of the oracle analysis was 8,436 milliseconds. The average CPU use was 9.73%, the physical memory use was 153,808 KB and the log file was 11.3 MB. These data indicate low resource usage. The analyzed time and resources support hypothesis H_1 from issue 4.

After the analysis, Apolom identified violations on rules $R3\_2$ and $R3\_6$. Figure 7.6 presents the oracle report.

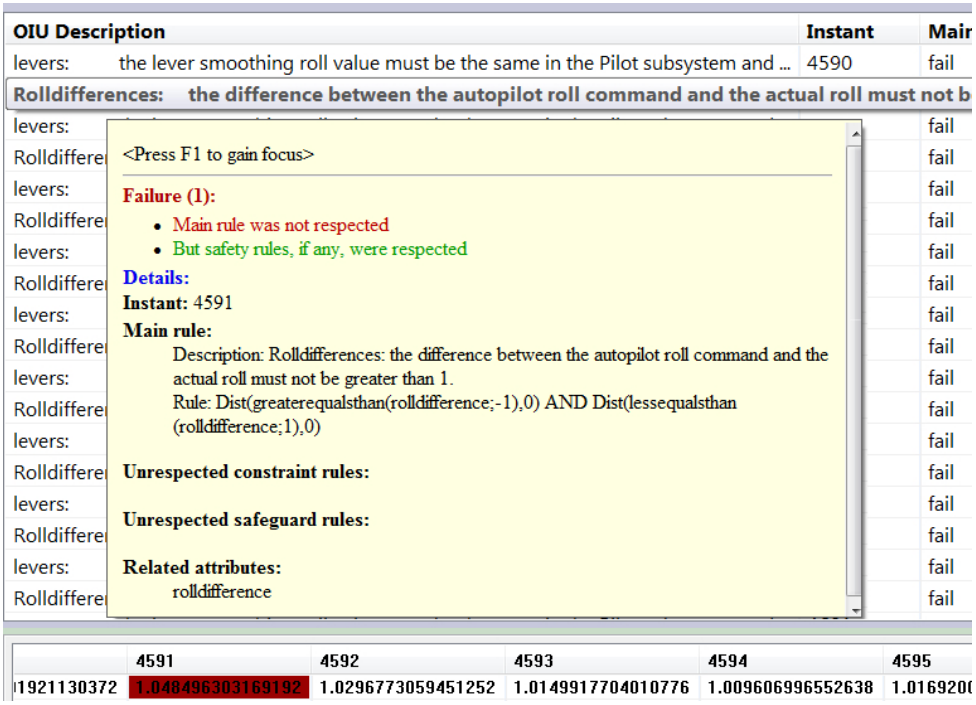


Figure 7.6: Oracle Report

In this example, the report shows that the main rule was violated, where the difference between the roll autopilot command and the actual UAV roll exceeds 1 degree (approximately 1.04849 degrees at instant 4591). Figure 7.6 also indicates that the safeguard rules were respected. In fact, the result showed that, even in the case of a system manually tested for several times and now in use, Apolom could find unrevealed errors.

7.3.6 Threats to Validity

An external threat to validity concerns to the subject. It is expected that the tester should have knowledge about the oracle specification language and the tool. The tool viability was also demonstrated, but the acceptance in the industry was not yet analyzed. As a next step, it is intended to present the tool to an aeronautics industry and measure its acceptance.

7.4 Evaluation 3

This section presents a study on the proposed off-line oracle access algorithm, *Fast Jumper* (Section 4.4.1). This algorithm was designed to improve accesses to a sequence of data in which values must be read in sequence, but other values which are not in sequence must also be read.

It may be a common scenario when temporal properties are evaluated. For example, let us consider a sequence which represents values over time. If simple temporal properties need to be checked as:

Verify if, in the next instant, A is greater than 10.

Then the analyzer may evaluate the expression ($A > 10$) for each next instant w.r.t the analysis instant, from the first to the last, in sequence. However, more complex temporal properties may demand that other values from the future or past must be analyzed at a given analysis instant, as:

If A is greater than 10 in the current instant, verify if it is also greater than 10 in the next 100000 instants, counting from the next 10000 instants

In such case, when an analyzer finds an instant in which $A > 10$ it must verify it again from the $10,000^{th}$ to the $110,000^{th}$ value in the sequence.

This study analyzes the following question: can *Fast Jumper* provide any advantage when compared with log reading without secondary memory spaces?

The independent variables are: Apolom tool, analyzed rules and Simulink models. The dependent variable is: the oracle analysis time and space.

7.4.1 Operation and Results

For this experiment, two models were used: Lorenz and UAV. In both cases, Fast Jumper was analyzed with different memory space sizes and with no secondary memory spaces.

Lorenz model: when the oracle analyzes the already described Lorenz specification, at some *analysis instant*, its procedure reads values in the current instant and in the next instant. For example, Equation 7.1 expresses an equation where *next* x (x_{i+1}) depends on the current x and y (x_i and y_i). Such equations may not be benefited by *Fast Jumper* because it only needs to read values in sequence, that is, all data are close enough to

fit in a main memory space, which is usually provided by a language built-in buffer, as *BufferedReader* or *BufferedInputStream* from Java.

The experiment with such rules evaluates whether the present proposal impacts negatively when the oracle analyzes rules that do not require a secondary memory space. The simulation was executed for 128,000 instants and analyzed with Apolom. Different configurations of Main Memory Spaces (MMS) and Secondary Memory Spaces (SMS) were used, by changing Apolom preset memory sizes.

This procedure was executed 32 times for each different memory space configuration. Table 7.2 summarizes the extracted data.

Table 7.2: Lorenz execution time

		Secondary Memory Space			
		none	**1024**	**2048**	**8192**
M. Memory Space	**none**	5347	678	640	619
	1024	706.6	618.2	619.1	599.3
	2048	666.6	611.9	613.5	594.8
	64000	606.9	586.7	596.2	586.5
	512000	605.4	595.1	595.5	596.4

When no memory space was used, the average analysis time was 5347ms (more than 9 times the best score). When using different combinations of memory spaces, the average did not vary more than 5.5% from the best to the worst time. Using two tail test with $\alpha = 0.05$ on the given results, many configurations presented statistical equivalency using the same MMS and different SMS:

- With no MMS: no result is statistically equivalent;
- With 1024 MMS: 1024 and 2048 SMS are statistically equivalents;
- With 2048 MMS: 1024 and 2048 SMS are statistically equivalents;
- With 64000 MMS: 1024 and 8192 SMS are statistically equivalents;
- With 512000 MMS: all results with SMS are statistically equivalents.

These results corroborate as expected: (i) when there is no MMS, then SMS is used and its size has impact over the time analysis; (ii) when the same MMS size is applied, the impact of SMS over rules which do not use them is small or virtually null (Figure 7.7).

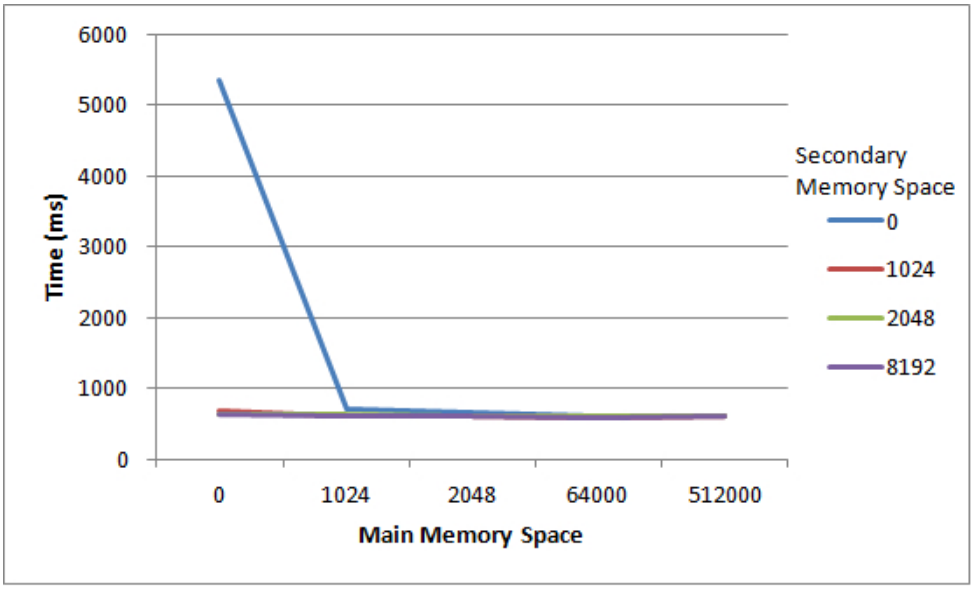


Figure 7.7: Relation of performances with different MMS and SMS

The results also indicates that MMS size affects the analysis time, but with small impact when considering the relation between MMS size and analysis time, as it can be observed in the blue line on Figure 7.7.

Next execution set evaluates the impact of a more costly operator in the analysis time:

Dist(greaterthan(Y;-100),0) ->

Lasting(greaterthan(Y;-100),1000,1500) AND Lasting(greaterthan(Y;-100),-1500,-1000)

The expression above uses *Lasting* operator. It verifies whether a behavior holds within an interval, which may be in the past and/or future. In this case the worst case scenario was forced, that is, all instants within the intervals holds, so the analyzer must verify all instants. Otherwise, if the analyzer detects that some instant does not hold within the interval, it would stop the analysis.

This rule differs from the first on the use of secondary memory spaces. The first rule does not use SMS because all required data are close enough to fit in the MMS. However, this second rule will require analysis in the past and future w.r.t. the analysis instant: 1000^{th} instant from the analysis instant to the 1500^{th} in the future and in the past, which will not fit in the MMS if its size is small. Table 7.3 presents the results.

The analysis takes approximately 8 minutes to be performed when no memory space is used. When only SMS is used, the analysis time drops from 31.82 up to 69.77 times.

Table 7.3: Fast Jumper: data interval behind and beyond main memory space

Main Memory Space (bytes)	**Secondary Memory Space** (bytes)	**Behavior analysis time** (ms)	**OIU analysis time** (ms)	**Total time** (ms)
none	none	3688	494102	497790
none	1024	636	7799	8435
none	2048	638	7701	8339
none	8192	608	6531	7134
1024	none	668	502834	503502
1024	1024	612	7795	8407
1024	2048	593	7845	8738
1024	8192	582	6422	7004
2048	none	586	18168	18754
2048	1024	593	7661	8254
2048	2048	587	7500	8087
2048	8192	587	6359	6946
64000	none	536	7255	7791
64000	1024	571	6000	6571
64000	2048	578	6012	6590
64000	8192	568	6022	6591
512000	none	568	6084	6652
512000	1024	562	6087	6649
512000	2048	571	5995	6566
512000	8192	576	6099	6675

Figure 7.8 illustrates the performance with different memory configurations. When MMS is not large enough to fit all data evolved in the expression and no SMS is provided, the analysis time takes 8 minutes. When the MMS is large enough to fit part or all data, the time drops drastically (Figure 7.8 (A)). Figure 7.8 (B) presents a more detailed relation between memory configurations when MMS encompass at least part of the data. It is possible to verify that better performances can be achieved with less memory using

SMS. When using large amounts of MMS (512,000 bytes), the SMS does not affect the result because MMS is large enough to fit all the data.

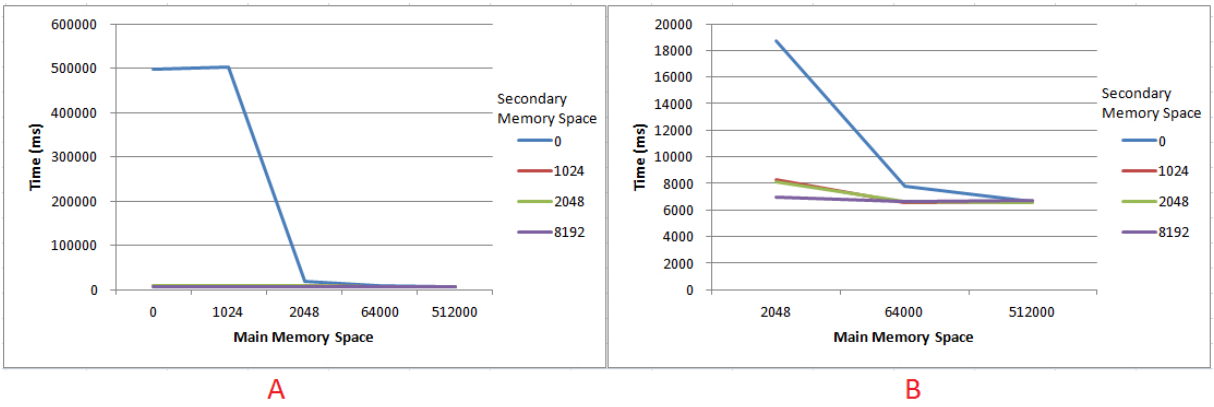


Figure 7.8: Relation of performances when SMS is used

It is worthy to note that all behavior analysis in this study will not use SMS if MMS is available. In this step, each equation will be analyzed w.r.t. (i) the analysis instant and (ii) next instant (Equations 7.1 to 7.3). It explains why *Behavior analysis* column does not present significant variations when SMS changes. However, as discussed before, if no MMS is available, then SMS will be used instead and time will drop, as it is showed in Table 7.3, from line two to five.

Only in the OIU analysis step, SMS will be allocated because the rule must be checked thousands of instants in the past and future.

Table 7.4 presents a summary of times when using large SMSs. It is possible to note that the difference of performances is not high. When the analysis uses only MMS (64000 and 512000 bytes), the best time is 6591ms. However, when SMS is used, the best time is 6946ms. This leads to the following question: "why not using large MMS and no SMS?". The presented example uses intervals of time 1000 instants from the analysis instant. Depending on the Simulink default configuration, such interval may represent 200 seconds or 20 seconds. A "real-world" requirement may demand intervals of much more than 1000 instants. In such cases, the MMS should be substantially greater than 512000 bytes to fit all analyzed data. However, with the use of SMS, less memory space needs to be provided with a comparable result.

For example, requirement $R2\_4$ from Section 7.3.4 demands analysis at 30000 instants in the future in a small trajectory of a UAV. Larger trajectories could take hours to the UAV pass by two waypoints and it would require too large MMS to fit all values from the analysis instant to the time in the future. It is possible to conclude that *Fast Jumper* has a suitable result with spend of small uses of memory spaces.

Table 7.4: Fast Jumper: large secondary memory space

Main Memory Space (bytes)	**Secondary Memory Space** (bytes)	**Behavior analysis time** (ms)	**OIU analysis time** (ms)	**Total time** (ms)
none	8192	608	6531	7134
1024	8192	582	6422	7004
2048	8192	587	6359	6946
64000	8192	568	6022	6591
512000	8192	576	6099	6675

When no SMS is used, analysis time is approximately 8.3 minutes. However, when SMS is used, time drops at least 56.97 times, which evidences that *Fast Jumper* provides the advantage of consuming less memory space and comparable time result when compared to large MMS-only usage. It also supports H_1 from question 4.

UAV model: this model was included in the experiment to provide results from a more complex and "real-world" model with a larger number of rules. Table 7.5 summarizes the results.

Table 7.5: Fast Jumper: a "real-world" study

		Secondary Memory Space			
		none	**1024**	**2048**	**8192**
M. M. Space	**1024**	3039	2819	2824	2812
	2048	3022	2849	2920	3036
	64000	2856	2865	2864	2869
	512000	2935	2939	2915	2930

There is no significant difference between the use of SMS because values outside MMS are seek only 16 times in the analysis, one for each time a Waypoint is reached and two when counting each Waypoint.

7.4.2 Threats to Validity

The results evidence that *Fast Jumper* may improve analysis time with different configurations and expressions, when values outside MMS are required. But a criterion to define when a MMS is considered impractically-large and when *Fast Jumper* is better suited has not been determined. Although such threshold is not yet defined, the proposed algorithm seems to have a better relation between memory use and analysis time than using only large MMS.

7.5 Evaluation 4

This study aims to answer question 5. Defects were introduced into copies of the original model and it was evaluated whether the oracle can identify any errors derived from such defects.

The independent variables are: Apolom tool, Simulink models, oracle specification and tester expertise. The dependent variables are: number of killed mutants, manual evaluation effectiveness, automated evaluation effectiveness and number of different detected failures.

7.5.1 Operation

Mutants were generated from both Lorenz and UAV models. Then, manual and automated effectiveness were compared by investigating the relation between the number of manual failure detection and automated failure detection.

The mutations were based on the operators proposed by Araujo et al. (2011a). Table 7.6 shows the selected operators for each model.

Table 7.6: Selected mutant operators

Model	**Operator**	**Description**
Lorenz	constant change	increases or decreases the value of constants of a model.
Lorenz and UAV	arithmetic op. replacement	replacements among blocks Add, Sub, Product, Divide and Gain.
UAV	statement swap	swaps first and third inputs of a Switch block,

The operators were selected by probabilistic sampling (two-stage sampling) as follows: (i) two of the five classes of operators were sorted; (ii) one operator from each selected class was sorted; (iii) eligible blocks in the original model were identified for each selected operator; (iv) ten blocks were sorted for mutation from the eligible group for each selected operator; (v) mutants were generated for each selected block for each operator, with a total of 20 mutants for each model.

Lorenz: the simulation of this model plots a graphic with the shape of a butterfly. Visually, it is expected a result as illustrated in Figure 7.9, from Stewart (1989).

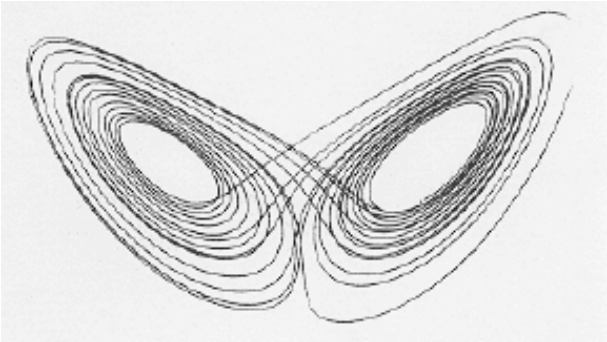


Figure 7.9: Lorenz Attractor example. Source: (Stewart, 1989)

This model was used to illustrate the difficulty in predict dynamical systems which are highly sensitive to initial conditions, as weather. Although the draw will have a butterfly pattern, its sequence of values may change widely if any initial condition is changed. Therefore, the visual evaluation is not of much use except to give an idea of its correct "butterfly" behavior.

UAV: the simulation of this model presents a result as shown in Figure 7.4 and explained in Section 7.3.1. The image presents a view of the passage of a UAV on four waypoints and its correction maneuvers. But evaluating the results manually is difficult. For example, the flight time between two waypoints may be laborious to assess because the tester must identify the instants in which the plane passes between one point and another and make the calculation. No facilities were provided to such manual analysis.

7.5.2 Results

Lorenz: first set of mutants (constant change operator) produced (i) 1 visibly wrong result; (ii) 1 suspect result; and, (iii) 8 visibly acceptable results. Figure 7.10 presents the original and 8 of 10 results.

Figure 7.10 (A) represents the original (correct) model. All other images are from the first mutant set. Image (B) is particularly similar to the original. Image (f) clearly presumes a failure and image (h) may raise suspicion. All other images may be considered

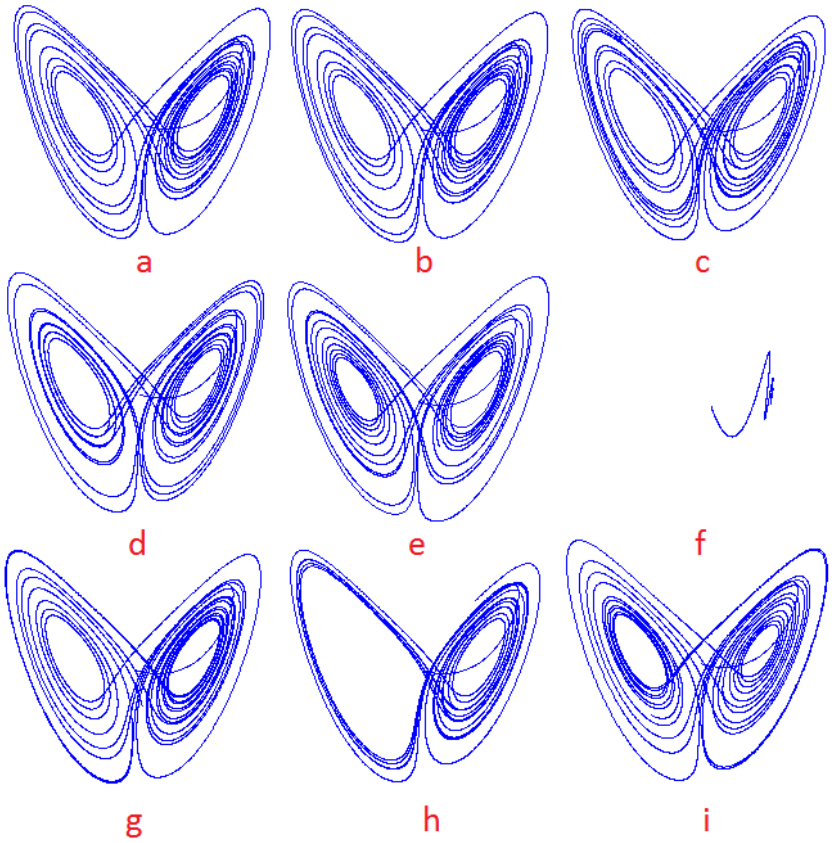


Figure 7.10: Lorenz Attractor samples.

viable solutions if no correct image (a golden version) is previously provided. However, Apolom found failures in all mutants.

Second set of mutants (arithmetic op. replacement operator) produced 9 visibly wrong results and 1 visibly acceptable result. Apolom found failure in all mutants.

It evidences that automated oracles may identify failures which would not be easily found manually, supporting statement H_1 from issue 5.

UAV: this model has a defect in relation to requirements $R3\_2$ and $R3\_6$ as reported in Section 7.3.5. First set of mutants (statement swap operator) produced (i) 2 models which could not be simulated, (ii) 4 simulations with visually detectable issues and (iii) 4 simulations with no apparent issues. Apolom detected problems in all cases w.r.t. requirements $R3\_2$ and $R3\_6$. In group (iii), despite the fact there are no visually detectable issues, Apolom identified rule violations in several instants, besides the already reported ones:

- The plane passes by all waypoints, but does not complete the route twice, as it was expected (failures on requirements $R2\_2 to R2\_6$);
- The plane took more than 90 seconds to pass between waypoint 2 to 3 (failure in requirement $R1\_8$);

- The difference between autopilot climb command and the actual climb must be less than 1, but it passed the threshold several times, up to 3.945 degrees (failure on requirement $R3\_9$);
- Airspeed value and airspeed difference between autopilot command and actual speed were slightly disrespected (failures in requirements $R3\_4$ and R_3_9);
- The difference between autopilot pitch command and actual pitch should not pass 1.5 degrees, but it passed up to 10 degrees (failure in requirement $R3\_7$).

The other 3 models did not affect the results w.r.t. the oracle specification.

Second set of mutants (arithmetic op. replacement operator) produced 9 results with no visually detectable issue and 1 result clearly wrong. From the former group, Apolom could find failures in 2 models:

- Mutant A:
  - Failure in reach waypoints in the expected time (Requirements $R2\_7$ and $R2\_8$)
  - Failure in complete the route as expected (Requirements $R2\_3$, $R2\_4$, $R2\_5$ and $R2\_6$);
  - The difference between the autopilot airspeed command and the actual airspeed should be less than 5m/s, but it exceeded 10m/s.
- Mutant B:
  - Vertical acceleration should be limited to $10m/s^2$, but it exceeded $84m/s^2$ (Requirement $R1\_4$);
  - The difference between the autopilot climb command and the actual climb should be less than 1, but it exceeded 1.6 (Requirement $R3\_9$).

Seven mutants were not caught by the oracle. A mutant had its block changed from *sum* to *subtract*. In this case, the output signals pass by a safeguard procedure in which climb values above or below a threshold are replaced by a standard climb value. Other mutant also had its operator block changed, but the blocks is unnecessary (or yet to be implemented) because it originally sums a value with 0 and the mutant subtracts the same value with 0 (in this case, they are equivalent). Five mutants are equivalent because their respective changes affect blocks in which their signals are not yet used in this version (disconnected lines).

Considering the mutant analysis, five of nine mutants are equivalent. From four non-equivalent mutants, one was not caught by the oracle and manual evaluation because

it passes by a safeguard procedure which normalizes abnormal values. Three mutants were identified by the oracle in which two of them were not identified by manual evaluation.

The results evidence that automated oracle analysis may detect failures which would be impossible or impractical to identify manually. It supports hypothesis H_1 from issue 5.

7.5.3 Threats to Validity

The oracle effectiveness may vary depending on how complete is the oracle specification, which is discussed in Section 3.3.1. A complete specification, as presented in Lorenz study reflects a highly effective oracle, detecting even many imperceptible failures by manual approach. A less complete specification, as presented in UAV study, may decrease the oracle effectiveness. However, it was also capable of detecting failures which would be difficult or impractical to find by manual means. Studies with more variety of models can provide a better generalization of the results.

7.6 Final Remarks

This chapter presented evidences which support the proposal viability. Five concerns were aimed. They are related to the oracle implementation feasibility, probe-effect, specification writing complexity, time and resource usage, and effectiveness.

Results from study 1 showed evidences that the proposed instrumentation may not affect the simulation time as it was implemented. It also supported that oracle specification may be simple.

Study 2 presented a "real-world" model with higher number of requirements. It evidenced that oracle specifications may be quickly written by a specialist and eventual rule complexities may be softened by mechanisms as the proposed "Rule Wizard".

Study 3 indicates that *Fast Jumper* may reduce analysis time, mainly when temporal operators require values from outside the main memory space, that is, beyond or behind the analysis instant vicinities.

Study 4 evidences that our proposal is capable to identify failures which could not be found manually, which supports its effectiveness.

The studies also show that an oracle generator tool is deployable. It also may help to soften the difficulty of writing formal specifications.

Chapter

8

Other Available Approaches

This chapter presents an overview of available tools with oracle support for Simulink. It aims to highlight the innovative part of this work, which is the use of an oracle generation approach in the context of Simulink models with temporal specification-based language.

8.1 Simulink Model Verification

Simulink provides a set of blocks to help the tester to check simple assertions. If an assertion failures, a warning is raised and the Matlab prompts the instants when it occurred.

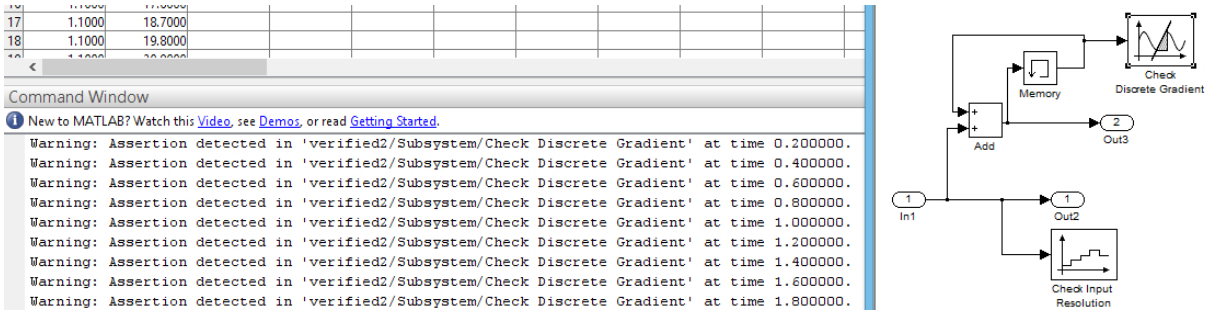


Figure 8.1: Simulink Model Verification

Figure 8.1 presents an example of a model (at the right) with two model verification blocks, *Check Input Resolution* and *Check Discrete Gradient*. The former asserts that

the signal must be equals to a constant value and the latter asserts that the difference between two consecutive samples of a signal must be less than a constant value. A list of warnings is available in the *Command Window*, which shows the failed assertion and the instants when it occurred.

For each Simulink Model Verification block assertion there is an equivalent behavior on the proposed approach. Table 8.1 presents the equivalence list.

Table 8.1: SMV blocks

Block	**Behavior**	**Description**
Check Dynamic Range	betweenS(seq1, seq2, seq3)	assert that a signal always lies between two other signals
Check Static Range	betweenC(seq; val1, val2)	assert that a signal is always less than a constant lower bound or greater than a constant upper bound
Check Dynamic Gap	gapS(seq1, seq2, seq3)	assert that a signal is always less than a lower bound signal and greater than a upper bound signal
Check Static Gap	gapC(seq; val1, val2)	assert that a signal is always less than a constant lower bound and greater than a constant upper bound
Check Dynamic Lower Bound	lessthanS(seq1, seq2)	assert that the input signal is less than another signal
Check Static Lower Bound	lessthanC(seq; val1)	assert that the input signal is less than a constant
Check Dynamic Upper Bound	greaterthanS(seq1, seq2)	assert that the input signal is greater than another signal
Check Static Upper Bound	greaterthanC(seq; val1)	assert that the input signal is greater than a constant
Check Input Resolution	equalsto(seq, <*resolution*>)	assert that the input signal has a specified resolution
Check Discrete Gradient	gradient(seq, <*maximum*>)	assert that the absolute value of the difference between successive samples of a discrete signal is less than an upper bound
Assertion	NOT equalsto(seq, 0)	asserts that an input is non-zero

The approach proposed in this thesis supports more expressive assertions (and an oracle procedure to analyze them). As instance, the tester may define in which intervals some condition should be checked. Furthermore, it allows the use of temporal quantifiers, which is not supported by Simulink Model Verification (SMV). Therefore, it is not possible to represent complex temporal properties with SMV, as the discussed Example 1 (Section 4.2.2, page **??**) and its rule representation presented in Section 4.2.1, page 46 (Example 2). In these examples, a behavior must be checked only if other behavior is found and there is a temporal property which relates both behaviors.

The proposed procedure also allows the detection of spurious occurrences, as it is discussed in Section 4.4.3. For example, a rule with *implies* connector denotes that if an expression A holds (left side of the implication rule) then another expression B must hold (right side of the implication). In this case, the proposed procedure analyzes whether B holds if A holds, but it is also capable of relating the occurrences of B with respect to the occurrences of A. Such feature allows the oracle procedure to identify when B holds and it is not related to any occurrences of A, that is, spurious occurrences of B.

Given a requirement as "if there is smoke, the fire alarm must be on", one may define a rule as:

$$detected(smoke) \implies on(alarm)$$

It may be required to certify that alarm is always off if no smoke is detected. The presented example is simple and a rule to identify such scenario could be easily defined as:

$$\neg detected(smoke) \implies off(alarm)$$

However, if alarm is on and no smoke is detected then this alarm state may be considered as a spurious occurrence of the first rule because it is an occurrence of *on(alarm)* which is not related to an occurrence of *detected(smoke)*. The proposed approach provides a procedure to detect spurious occurrences (Section 4.4.3) without the need to write an additional rule, as the second expression.

It is worthwhile to note that defining rules to identify spurious occurrences become more complex as more elaborated is the rule, as the example presented in Section 4.2.1 (page 46). In this example, analysis of temporal properties is required which makes difficult the writing of a spurious detection rule. In fact, it may be impossible to determine exactly which occurrences are spurious without an assumption, as it is discussed in Section 4.4.3 (page 56) and exemplified in Section 5.3 (page 71) and Section 6.3.1 (page 81).

The spurious detection is also a contribution to alleviate the burden of writing complex assertions and Simulink Model Verification does not provide such mechanism.

Another novelty w.r.t. SMV is the trigger-dependency mechanism present within the OIUs. Its description is presented in Section 4.2.2, page 47, and examples of its application is found in Section 5.4, page 73 and Section 7.3.4, page 100 ($R3\_6_{s1}$ and $R3\_6_{s2}$).

The proposed approach allows modularized organization of its requirements. And, finally, it enables one to implement a tool, as Apolom, with more detailed report of the failed assertions (rules), with optional natural language description of each rule, the respective failure instants, the related sequences and their values in the vicinity of such instants (Figure 8.2).

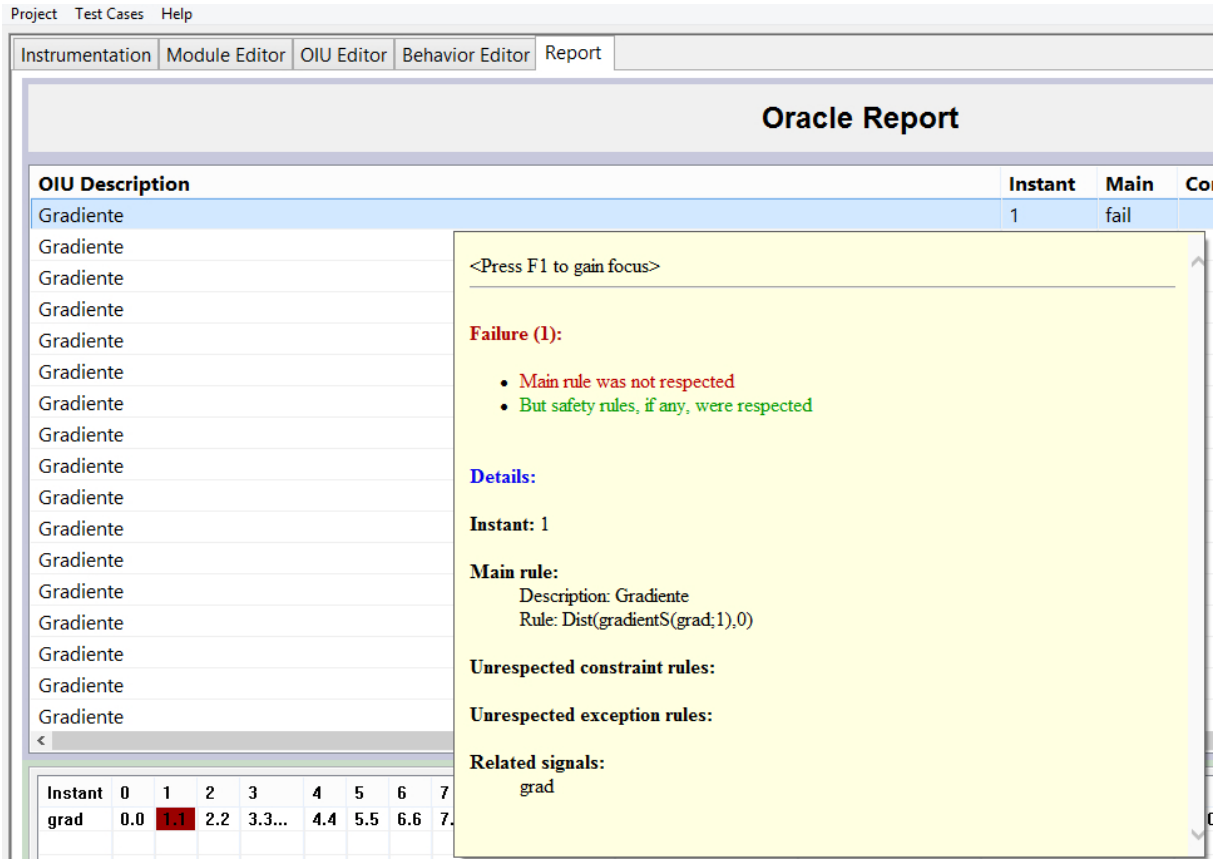


Figure 8.2: Apolom report

The contributions of this work with respect to the Simulink Model Verification library are here summarized: (i) **higher expressiveness** with quantitative and qualitative temporal operators, existential and universal quantifiers and an oracle procedure to interpret such expressions; (ii) **detection of spurious occurrences**, which reduces the need to write rules; (iii) **oracle assumptions** that allow the tester to decide how the procedure will analyze temporal properties; (iv) **trigger-dependency between rules**, which allows the identification of different types of violation and the capability to express how

a simulation should behave in case some property is violated, an limitation discussed in Section 3.3.1, page 28; (v) **modular organization** of the oracle information; (vi) a **dedicated oracle environment** which separates both concerns, simulation and testing. An advantage of such contribution is that the tester may plan this step and write the specification independently of the model until it is ready for testing; (vii) **changing of paradigm**: a Simulink subsystem could be planned to represent a requirement, thus, working as an oracle. But the changing of paradigm gives higher confidence that a same error is not present both in the original model and oracle (Brown et al., 1992), as discussed in Chapter 2.2.1; (viii) a more **detailed report**.

8.2 Simulink Verification and Validation

Simulink Verification and Validation (SVV) is a toolbox from MathWorks which automates requirements tracing, model coverage analysis, and modeling standards compliance checking.

The checking is performed by the Model Advisor, which verify a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation, such as unconnected lines, disabled or unresolved library links and optimization settings.

The model coverage analysis checks for metrics as: cyclomatic complexity, decision, signal range and condition coverage.

SVV provides a library with a System Requirement block. It lists all the system requirements associated with the model or subsystem depicted in the current diagram. The tester must link each requirement with its respective representation in a document and with objects in the model. It allows the tester to navigate between the model and the specification.

This toolbox may be considered as a complementary solution to the present approach. It provides model coverage and compliance checking which the present approach does not support. SVV also encompass tracing capabilities.

The contributions of this work with respect to the Simulink Verification and Validation toolbox are the same discussed in Section 8.1.

8.3 Simulink Design Verifier

This product from MathWorks provides formal verification on Simulink models. With static code analysis, it detects and proves the absence of overflow, divide-by-zero, out-of-bounds array access, and other run-time errors. Simulink Design Verifier (SDV) also generates a counter example if a property can not be proved.

The properties are written with one of five blocks from SDV library. As example, given by MathWorks, one may want to prove that a signal is always less or equals to 0 (Figure 8.3).

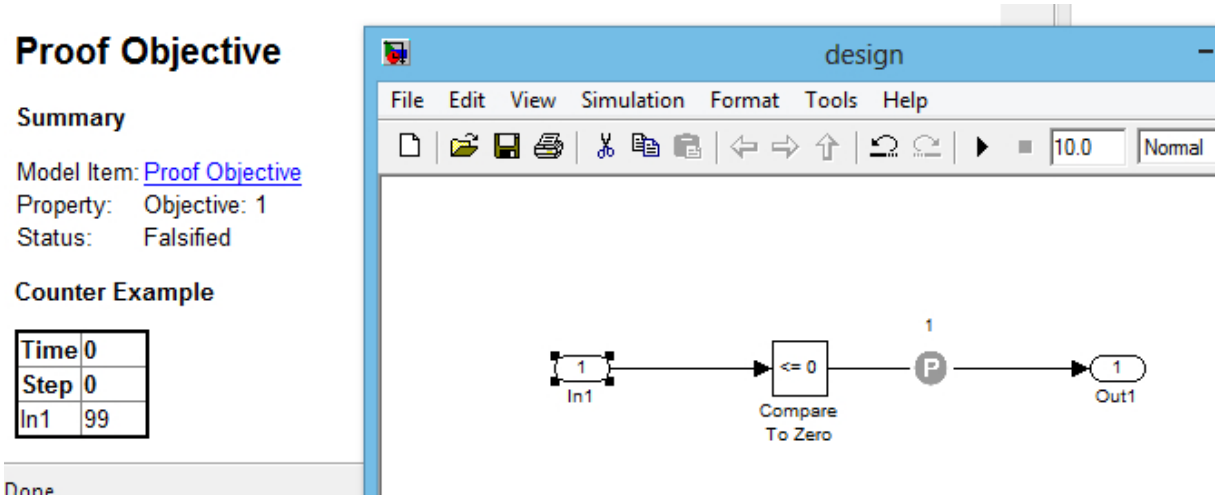


Figure 8.3: Simulink Design Verifier

Compare to Zero block generates 1 if the respective condition is true or 0 otherwise. *P* is a *Proof Objective* block from SDV library, which defines a property that a signal must satisfy. Since the objective is to prove that the input signal is always less or equals to 0 then the signal from the comparison block must always be 1 (true). Therefore, *P* block is set as 1.

The tester starts the proof verification from the *tool* menu and the algorithm tries to generate a counter example that prevents the proof. Figure 8.3 shows a counter example which demonstrates that the required property will not be satisfied if the input signal is equals to 99.

On the other hand, if the input block is replaced by a constant block which generates always 0, the analysis result will not find any counter example, since it is guaranteed that the signal will never be different of 0.

Figure 8.4 presents a valid proof that the property will be always respected.

However, this approach has limitations: the Design Verifier does not analyze models which do not represent a single output function, as the Lorenz model; It only analyzes models configured to run as fixed-step; An error was also found during the analysis, when replacing the *input* block for other source blocks as *Ramp* (Figure 8.5).

Error code 8 represents an internal error in the Simulink Design Verifier analysis engine. The product support asks to contact MathWorks and send the model for analysis.

This work focuses on the oracle problem, a specific domain of software testing, which differs from formal method by the static nature of the latter. Both are complementary and the merits of their comparison are not approached.

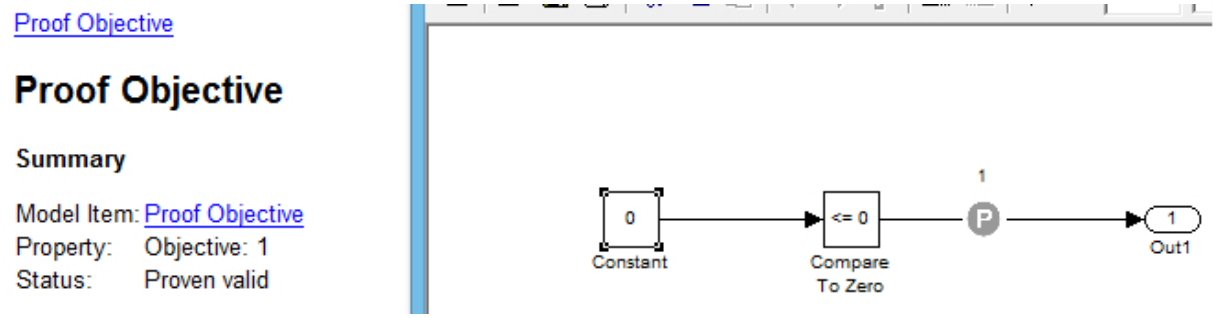


Figure 8.4: A valid proof

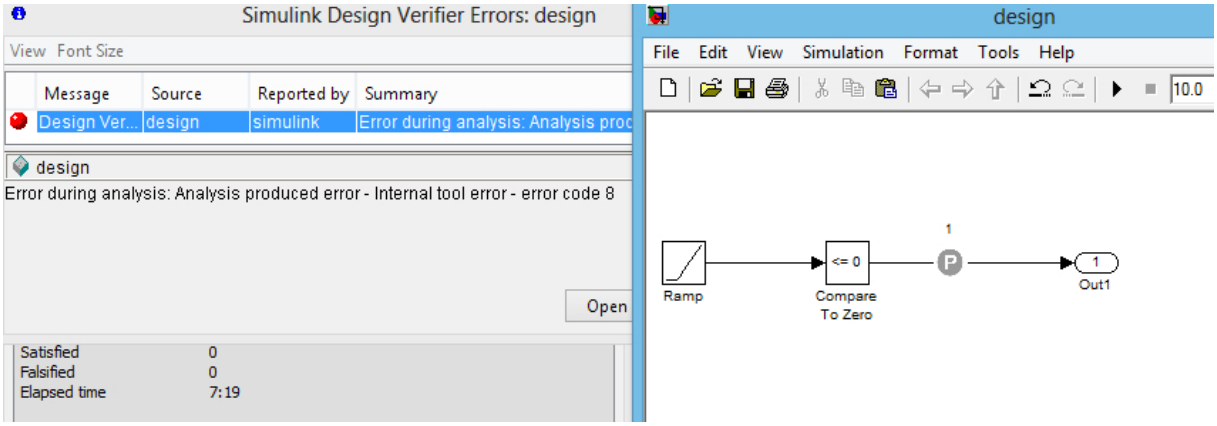


Figure 8.5: A Design Verifier internal error

The contributions of this work with respect to Simulink Design Verifier are also the same discussed in Section 8.1.

8.4 REACTIS

Reactis is a commercial tool which can be used to validate Simulink models and testing for its conformance. It is composed by three main components: simulator, tester and validator.

Reactis Simulator provides means to model debug, as breakpoints, single-step execution and coverage tracking. Reactis Tester automatically generates test cases from models which can be used to check conformance between model and implementation. Reactis Validator checks for violations of user-defined assertions.

Such assertions may be defined as C functions or state machines (the latter, with a Matlab environment called stateflow). An example of assertion is given in its user's guide (Reactive Systems, 2012):

When active, cruise control shall not permit actual, desired speeds to differ by more than 1 mph for more than 3 seconds.

A state machine definition is given in Figure 8.6.

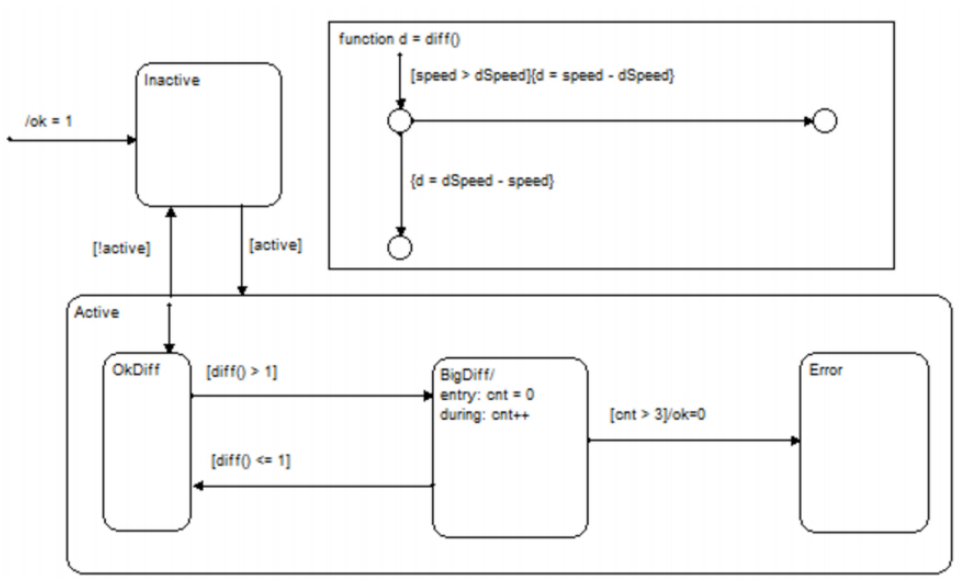


Figure 8.6: Reactis stateflow requirement. Source: Reactive Systems (2012)

Another assertion representation, written in C, is given in Figure 8.7.

```
143  int cruise_assert_spdCheck(int active,
144                             double speed, double dSpeed){
145    static int state = SPDCHECK_NOTINIT, cnt;
146
147    switch( state )
148      {
149      case SPDCHECK_NOTINIT:
150        {
151          /* Stateflow chart starts out not initialized,
152          first step is taken up with initializing it */
153          state = SPDCHECK_INACTIVE;
154          break;
155        }
156
157      case SPDCHECK_INACTIVE:
158        {
159          if( active ) state = SPDCHECK_ACTIVE_OK;
160          break;
161        }
162
163      case SPDCHECK_ACTIVE_OK:
164        {
165          if( !active )
166            state = SPDCHECK_INACTIVE;
167          else if( fabs(speed-dSpeed) > 1 )
168            {
169              state = SPDCHECK_ACTIVE_BIGDIFF;
170              cnt   = 0;
171            }
172          break;
173        }
174
175      case SPDCHECK_ACTIVE_BIGDIFF:
176        {
177          if( !active )
178            state = SPDCHECK_INACTIVE;
179          else if( cnt>3 )
180            state = SPDCHECK_ACTIVE_ERROR;
181          else if( fabs(speed-dSpeed) <= 1 )
182            state = SPDCHECK_ACTIVE_OK;
183
184          cnt++;
185          break;
186        }
187
188      case SPDCHECK_ACTIVE_ERROR:
189        {
190          if( !active )
191            state = SPDCHECK_INACTIVE;
192          break;
193        }
194      }
195
196    return (state!=SPDCHECK_ACTIVE_ERROR);
197  }
```

Figure 8.7: Reactis stateflow requirement. Source: Reactive Systems (2012)

The same assertion, in TRIO/Apolom is written as:

Starts(equals(speedcheck,1)) AND h=nowOn(equals(speedcheck,1))->

NOT Exists(Lasts(diffG(speed,dSpeed;1),14),0,h-16)

The complexity of the TRIO/Apolom expression is notably simpler than stateflow and C examples.

Reactis has a better implementation of model presentation than Apolom. However, it does not provide means to structure the specification, as modularization and trigger-dependent rules. Also, it does not provide algorithms to deal with different analysis assumptions or expressive temporal property evaluation and detection of spurious occurrences.

The contributions of this work with respect to Reactis are the same discussed in Section 8.1. In addition, Reactis does not seem to support extensibility to other tools as Scicos or XCos and it is not free.

8.5 T-VEC

T-VEC is an integrated development environment and associated specification and verification method (Blackburn and Busser, 1996). The tool allows the generation of test inputs, expected outputs, and a mapping of each test to the associated requirement.

As an oracle, it can detect computation errors and domain errors within the specification. The former occurs when the correct path through the program is taken, but the output is incorrect due to faults in the computation along the path (Howden, 1976; Zeil, 1989). A domain error occurs when an incorrect output is generated due to executing the wrong path through a program (Howden, 1976).

Other important concepts associated with T-Vec are:

- Ground term: an input variable used in a relation with a constant
- Ground clause: a variable and/or function used in a relation with a constant
- Clause: a relation between any combination of two or more input variables and/or functions

A test vector generation example is given by Blackburn and Busser (1996) with the following specification fragment:

$x \geq 5 \wedge$

$x + y \geq 6 \wedge$

$x - y \leq z \wedge$

$sin(z) \geq 0.5$

T-Vec initially identify the initial domain for each variable. Then, it limits such domains based on all ground terms. For example, it analyzes first line (which is a ground term) and limits the low bound of x domain to 5. Next step limits the domains based on ground clauses, in which the low bound of y domain is set to 1, so it can satisfy second line from the specification fragment. It also limits low bound of z based on $sin(z) \geq 0.5$. The analyzer defines the domain limits for all the other clauses. Using low and high bounds, as heuristic methods to select other values within the domain, the inputs are selected and expected outputs are calculated for the output variables. Such text vector can then be applied to an implementation or a Simulink-model.

T-Vec is also capable of identify specification inconsistencies: if it is impossible to identify a subdomain for a variable, a specification inconsistency is detected. For example, it would be impossible to identify a subdomain for z if the third clause had a *multiply* operator instead of *subtract* because $x * y$ would have a subdomain low bound of 5 ($5 * 1$), which is outside z domain. Therefore, z could never be greater than 5, which is a specification inconsistency.

The contributions of this work with respect to T-Vec are the same discussed in Section 8.1 and Section 8.4, except for modularization, which is also provided by this tool.

8.6 Considerations

This chapter presented an overview of available approaches with some level of oracle support for Simulink and highlighted the innovative part of this work in relation to them, which are: higher expressiveness, detection of spurious occurrences, oracle assumptions, trigger-dependency between rules, modular organization (except for T-Vec), dedicated oracle environment (except for REACTIS and T-Vec), changing of paradigm and detailed report on simulation failures with respect to the specification.

T-Vec and Reactis are well stated commercial products developed by large teams, and it is not intended to diminish their qualities and importance in the industry. However, the proposed oracle procedure supports temporal language analysis, detection of spurious occurrence, different analysis assumptions and the specification writing is softened by a Rule Wizard. Also, the proposed approach is presented in details and Apolom is available as a free and open source tool.

Chapter

9

Conclusions and Final Considerations

The importance of the testing activity is widely known. Several techniques and criteria have been proposed in order to find errors and increase the confidence over a system in development. Nevertheless, testing is not trivial and if part of the task is not automated, the cost of such activity may be high given the time restrictions and potential human mistakes. Therefore, support tools are essential to increase the quality of final systems by providing more rigor and precision and by eliminating error-prone manual analysis. Furthermore, it can reduce the costs associated with testing, and they would also foster a gain in productivity.

The development of embedded systems – as avionics, automotive and telco – are usually supported by modeling, analysis and simulation tools which allow the developer to define and evaluate a system behavior before it is implemented and deployed in its target hardware. Simulink, as other similar tools, is a de-facto standard for design and simulation of embedded systems in many different domains.

The simulation of a modeled system may require and produce, respectively, a large number of input and output. It hampers an adequate data selection and output evaluation.

This thesis addresses the oracle problem. Usually, the oracle role is played by a human tester. This research contributes in the definition of an approach which is a foundation to an engineering, allower of oracle generator automation.

An iterative oracle process provides the overview of three basic activities of an oracle generation. It includes the definition of its information, the mapping between such information and the original model under test, and the definition of the analysis assumption which implicates how the oracle should analyze the sequences produced in the simulation. The process foresees the incremental definition of the information by providing means to modify it without impacting the result consistency.

The oracle information must be formal enough to be analyzed by automated means. In this sense, this thesis also provides a method to specify the requirements of interest which can be interpreted by an oracle procedure. Although our proposal is language-independent, TRIO is here considered as a language of choice given its high expressiveness and easy syntax.

Also, the mapping between model and specification is a fundamental part of the process. Simulink-like tools have libraries with blocks which represent different types of output. But relying only on such blocks would allow only system and interface testing. Unit testing and a larger variety of integration testing can be guaranteed by providing the mapping between the specification and any signal from the model.

Since the oracle problem resides in the identification of a procedure to decide whether the obtained result matches the expected result, it is essential that the oracle is able to acknowledge the requirement and analyze the output properly. This thesis proposes a procedure and assumptions to analyze the results.

The availability of a tool which automates part of the oracle definition and the analysis granted an empirical evaluation of the thesis. Four studies with Apolom, a tool based on this research, showed that applying the proposed solution is viable and can actually reduce the time to identify errors and increase the potential to find them.

First study demonstrated that the probe-effect is not an issue for the proposed solution of mapping. It also evidences that the oracle specification may be relatively simple to elaborate.

Second study was executed with a large-scale real-world model. It supported the evidence that a specification can be simple to elaborate, as well as the time and cost are acceptable. Lastly, it showed that an automated oracle may be an efficient way to find uncovered errors.

Third study indicated that *Fast Jumper* is a viable solution to access large amounts of sequential and not sequential values for the analysis of temporal properties.

Fourth study resorted on mutation test to indicate the efficiency in finding errors. It evidences that an automated oracle may find errors which could not be practical by manual means.

9.1 Contributions

The thesis first contribution is a technical report on test oracles (Nardi and Delamaro, 2011), which may be used as a basis to other researches on specification-based oracles, as well as other oracle information paradigms, whereas it encompasses oracle studies with machine learning, metamorphic relations and n-version-*like* oracles.

A foundation to oracle engineering was defined, providing automation capability with high expressiveness to Simulink-like models. It overcomes limitations of productivity as error-prone comparisons, time and resource issues (Chapter 7). The process model (Chapter 4) guides a tester on the steps to define an oracle; the information definition method (Section 4.2) provides an structured and easy approach which may be applied to executable specification languages, therefore, allowing automated comparison between specification and simulation. The basic structure, the OIU, provides a dependency mechanism between rules which grants different levels of failure reports, requirement design and softens a common issue pointed by Nadeem and Jaffar-ur Rehman (2005) on the study of automated oracles: in this context specifications usually do not describe what a system needs to do when invalid input is given. Modules allow the organization of requirements into a tree structure, providing a comprehensive arrangement of OIUs; the procedure (Chapter 4.4) encloses algorithms that provide different comparison assumptions, which may soften the burden of writing a requirement.

The proposed solution extensibility is twofold: it is language-independent, which allows the inclusion of other specification languages, and it is easily extensible to other similar tools as XCos and SCicos.

A tool (Section 5) was implemented to allow the thesis evaluation. It is capable of analyzing Simulink models and may serve as a prototype to oracle generation tools.

9.2 Limitations

The definition of the oracle engineering was based primarily on the study of gaps in other approaches (Chapter 3 and Chapter 8) and on interview with development teams from the industry, namely: AGX and Embraer teams. Although it covers a large range of academic studies, the generalization of our approach is threatened by other needs in the industry, besides the ones considered from the quoted teams. Thus, as long as more needs are gathered from industry or researchers, it is planned to re-evaluate the presented solution to enhance it, if needed.

The main features of this approach were tested in several models; however, the evaluation of this thesis was based on two models. Even though the second and third studies

were performed with a real-world model of large-scale and real requirements, the accuracy of the results is threatened by the small variety of domains.

Apolom is a tool designed and implemented by one person for the period of one year. Its limitations are mostly regarded to time restriction and programming skills.

If a model is changed during the testing activity, it should not cause consequences to the oracle, except if a mapped line is removed. To avoid instrumentation errors, a tool should check if mapped lines were removed from the original model and alert the tester.

We consider that all described limitations can be overcome and it does not invalidate the proposed approach.

9.3 Possible Research Directions

As possible activities to be continued from the present thesis, more experiments in different domains are highlighted to improve the generalization of the results.

It is intended to explore the tool usability with an improved Rule Wizard. In interviews, it was one of the most noted features. When the interviewed testers were asked about obstacles to employ an automated oracle approach, many considered the cost to learn a specification language as the preponderant drawback.

Another possible improvement in the tool is extending the language with more built-in functions. Also, the support to XCos and Scicos, as the inclusion of other languages may improve the acceptance in the academy and industry.

Although the oracle allows trace between the specification and informal written requirements, it does not support trace with outside documents. Such direction is an interesting complement to the oracle automation.

Finally, other oracle information sources are complementary to the specification-based paradigm. Researches on the integration between different types of oracle may prove to be a challenging, yet promising field.

Bibliography

Agarwal, D. *A comparative study of artificial neural networks and info fuzzy networks on their use in software testing.* Doctoral Dissertation, University of South Florida, 2004.

Agarwal, D.; Tamir, D.; Last, M.; Kandel, A. A comparative study of artificial neural networks and info-fuzzy networks as automated oracles in software testing. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, v. 42, n. 5, p. 1183 –1193, 2012.

Aggarwal, K. K.; Singh, Y.; Kaur, A.; Sangwan, O. P. A neural net based approach to test oracle. *SIGSOFT Softw. Eng. Notes*, v. 29, n. 3, p. 1–6, 2004.

Aichernig, B. Automated black-box testing with abstract vdm oracle. In: *Computer Safety, Reliability and Security*, v. 1698 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 250–259, 1999.
Available on: `http://dx.doi.org/10.1007/3-540-48249-0_22`

Aichernig, B. K.; Griesmayer, A.; Johnsen, E. B.; Schlatte, R.; Stam, A. Conformance testing of distributed concurrent systems with executable designs. *Formal Methods for Components and Objects: 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, p. 61–81, 2009.

Alawneh, S.; Peters, D. Specification-based test oracles with junit. In: *23rd Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2010, p. 1–7.

Alligood, K. T.; Sauer, T. D.; Yorke, J. A. *Chaos: an introduction to dynamical systems.* New York: Springer-Verlag, 2000.

ALMOG, D.; HEART, T. Developing the basic verification action (bva) structure towards test oracle automation. In: *International Conference on Computational Intelligence and Software Engineering (CiSE)*, 2010, p. 1 –4.

ALPAYDIN, E. *Introduction to machine learning.* 2nd ed. The MIT Press, 2010.

ANDREWS, J.; FU, R.; LIU, V. Adding value to formal test oracles. In: *ASE 2002: Proceedings of the 17th IEEE International Conference on Automated Software Engineering.*, 2002, p. 275–278.

ANDREWS, J.; ZHANG, Y. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, v. 29, n. 7, p. 634–648, 2003.

ANTOY, S.; HAMLET, D. Self-checking against formal specifications. In: *ICCI '92: Proceedings of the Fourth International Conference on Computing and Information.*, 1992, p. 355–360.

ANTOY, S.; HAMLET, D. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, v. 26, n. 1, p. 55–69, 2000.

ARAUJO, R. F.; VINCENZI, A. M. R.; DELEBECQUE, F.; MALDONADO, J. C.; DELAMARO, M. E. Devising mutant operators for dynamic systems models by applying the hazop study. In: *Proceeding of the Sixth International Conference on Software Engineering Advances*, 2011a, p. 58–64.

ARAUJO, W.; BRIAND, L.; LABICHE, Y. On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM), 2011*, 2011b, p. 10 –19.

BAGGE, A.; HAVERAAEN, M. Axiom-based transformations: Optimisation and testing. *Electron. Notes Theor. Comput. Sci.*, 2009.

BAHAROM, S.; SHUKUR, Z. The conceptual design of module documentation based testing tool. *Journal of Computer Science*, v. 4, n. 6, p. 454–462, cited By (since 1996) 1, 2008.

BAHAROM, S.; SHUKUR, Z. Utilizing an abstraction relation document in grey-box testing approach. In: *ICEEI '09: International Conference on Electrical Engineering and Informatics.*, 2009, p. 304–308.

BAHAROM, S.; SHUKUR, Z. An experimental assessment of module documentation-based testing. *Information and Software Technology*, v. 53, n. 7,

p. 747–760, cited By (since 1996) 1, 2011.
Available on: http://www.scopus.com/inward/record.url?eid=2-s2.0-79955065276&partnerID=40&md5=a2df94b2348a733997f602a106f8519c

Bai, X.; Hou, K.; Lu, H.; Zhang, Y.; Hu, L.; Ye, H. Semantic-based test oracles. In: *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, 2011, p. 640 –649.

Barbosa, A.; Paiva, A. C.; Campos, J. C. Test case generation from mutated task models. In: *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, New York, NY, USA: ACM, 2011, p. 175–184 (*EICS '11*,).
Available on: http://doi.acm.org/10.1145/1996461.1996516

Baresi, L.; Bianculli, D.; Guinea, S.; Spoletini, P. Keep it small, keep it real: Efficient run-time verification of web service compositions. In: Lee, D.; Lopes, A.; Poetzsch-Heffter, A., eds. *Formal Techniques for Distributed Systems*, v. 5522 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 26–40, 2009.
Available on: http://dx.doi.org/10.1007/978-3-642-02138-1_2

Baresi, L.; Young, M. *Test oracles.* Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., http://www.cs.uoregon.edu/~michal/pubs/oracles.html, 2001.

Bennani, S.; Looye, G. H. N. Design of flight control laws for a civil aircraft using μ-synthesis. 1998.

Bieman, J.; Yin, H. Designing for software testability using automated oracles. In: *Test Conference, 1992. Proceedings., International*, 1992, p. 900–.

Biolchini, J. C. d. A.; Mian, P. G.; Natali, A. C. C.; Conte, T. U.; Travassos, G. H. Scientific research ontology to support systematic review in software engineering. *Adv. Eng. Inform.*, v. 21, n. 2, p. 133–151, 2007.

Blackburn, M. R.; Busser, R. D. T-vec: A tool for developing critical systems. In: *In Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96*, IEEE Computer Society Press, 1996, p. 237–249.

Bouchet, J.; Madani, L.; Nigay, L.; Oriat, C.; Parissis, I. Formal testing of multimodal interactive systems. *Engineering Interactive Systems: EIS 2007 Joint Working Conferences, EHCI 2007, DSV-IS 2007, HCSE 2007, Salamanca, Spain, March 22-24, 2007. Selected Papers*, p. 36–52, 2008.

Branco, K.; Pelizzoni, J.; Oliveira Neris, L.; Trindade, O.; Osorio, F.; Wolf, D. Tiriba - a new approach of uav based on model driven development and multiprocessors. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011, p. 1 –4.

Briand, L.; Labiche, Y. A uml-based approach to system testing. 2001. Available on: http://dx.doi.org/10.1007/3-540-45441-1_15

Briand, L.; Labiche, Y. A uml-based approach to system testing. *Software and Systems Modeling*, v. 1, n. 1, p. 10–42, 2002. Available on: http://dx.doi.org/10.1007/s10270-002-0004-8

Briand, L.; Labiche, Y.; Sun, H. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Software - Practice and Experience*, v. 33, n. 7, p. 637–672, cited By (since 1996) 14, 2003.

Brown, D.; Roggio, R.; Cross, J.H., I.; McCreary, C. An automated oracle for software testing. *IEEE Transactions on Reliability*, v. 41, n. 2, p. 272–280, 1992.

Chan, W.; Chen, T.; Cheung, S.; Tse, T.; Zhang, Z. Towards the testing of power-aware software applications for wireless sensor networks. 2007a. Available on: http://dx.doi.org/10.1007/978-3-540-73230-3_7

Chan, W.; Ho, J.; Tse, T. Piping classification to metamorphic testing: An empirical study towards better effectiveness for the identification of failures in mesh simplification programs. In: *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, 2007b, p. 397–404.

Chan, W.; Tse, T. Oracles are hardly attain'd, and hardly understood: confessions of software testing researchers. In: *The Symposium on Engineering Test Harness (TSETH '13), in Proceedings of the 13th International Conference on Quality Software (QSIC '13), IEEE Computer Society, Los Alamitos, CA*, 2013.

Chan, W. K.; Cheng, M. Y.; Cheung, S. C.; Tse, T. H. Automatic goal-oriented classification of failure behaviors for testing xml-based multimedia software applications: an experimental case study. *J. Syst. Softw.*, v. 79, n. 5, p. 602–612, 2006.

Chapoutot, A.; Martel, M. Abstract simulation: A static analysis of simulink models. In: *ICESS '09: International Conference on Embedded Software and Systems.*, 2009, p. 83 –92.

Chen, J.; Aoki, T. Conformance testing for osek/vdx operating system using model checking. In: *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, 2011, p. 274 –281.

Chen, J.a, S. S. b. Specification-based testing for gui-based applications. *Software Quality Journal*, v. 10, n. 3, p. 205–224, cited By (since 1996) 10, 2002.

Chen, T.; Feng, J.; Tse, T. Metamophic testing of programs on partial differential equations: A case study. In: *Proceedings - IEEE Computer Society's International Computer Software and Applications Conference*, Oxford, 2002, p. 327–333.

Chen, T.; Kuo, F.-C.; Tse, T.; Zhou, Z. Q. Metamorphic testing and beyond. In: *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on*, 2003, p. 94–100.

Chen, T.; Tse, T.; Zhou, Z. Fault-based testing in the absence of an oracle. In: *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, 2001a, p. 172 –178.

Chen, T.Y.a, T. T. Z. Z. Fault-based testing without the need of oracles. *Information and Software Technology*, v. 45, n. 1, p. 1–9, cited By (since 1996) 16, 2003.

Chen, T. Y.; Cheung, S. C.; Yiu, S. M. Metamorphic testing: a new approach for generating next test cases. *asa*, 1998.

Chen, T. Y.; Tse, T. H.; Zhou, Z. Fault-based testing in the absence of an oracle. In: *Proceedings - IEEE Computer Society's International Computer Software and Applications Conference*, Chicago, IL, 2001b, p. 172–178.

Cheon, Y. Abstraction in assertion-based test oracles. In: *Quality Software, 2007. QSIC '07. Seventh International Conference on*, 2007, p. 410–414.

Cheon, Y.; Avila, C. Automating java program testing using ocl and aspectj. In: *Seventh International Conference on Information Technology: New Generations (ITNG).*, 2010, p. 1020 –1025.

Cheon, Y.; Leavens, G. A simple and practical approach to unit testing: The jml and junit way. In: *ECOOP 2002 Object-Oriented Programming*, 2002, p. 1789–1901.

Cho, S. M.; Lee, J. W. Lightweight specification-based testing of memory cards: A case study. *Electron. Notes Theor. Comput. Sci.*, v. 111, p. 73–91, 2005.

Coppit, D.; Haddox-Schatz, J. On the use of specification-based assertions as test oracles. In: *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, 2005, p. 305–314.

Dan, L.; Aichernig, B. K. Combining algebraic and model-based test case generation. 2005.
Available on: `http://www.springerlink.com/content/qnfpx7hkqphryx3l`

Davis, M. D.; Weyuker, E. J. Pseudo-oracles for non-testable programs. In: *ACM '81: Proceedings of the ACM '81 conference*, New York, NY, USA: ACM, 1981, p. 254–257.

Ding, J.; Wu, T.; Lu, J.; Hu, X.-H. Self-checked metamorphic testing of an image processing program. In: *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI).*, 2010, p. 190 –197.

Ding, J.; Wu, T.; Wu, D.; Lu, J. Q.; Hu, X.-H. Metamorphic testing of a monte carlo modeling program. In: *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, New York, NY, USA: ACM, 2011, p. 1–7 (*AST '11*,).
Available on: `http://doi.acm.org/10.1145/1982595.1982597`

D'Souza, D.; Gopinathan, M. Computing complete test graphs for hierarchical systems. In: *SEFM'06: Fourth IEEE International Conference on Software Engineering and Formal Methods.*, 2006, p. 70–79.

Durrieu, G.; Waeselynck, H.; Wiels, V. Leto - a lutre-based test oracle for airbus critical systems. *Formal Methods for Industrial Critical Systems: 13th International Workshop, FMICS 2008*, 2008.

Eckhardt, D.E., J.; Lee, L. D. A theoretical basis for the analysis of multiversion software subject to coincident errors. *Software Engineering, IEEE Transactions on*, v. SE-11, n. 12, p. 1511–1517, 1985.

Edwards, S. A framework for practical, automated black-box testing of component-based software. *Software Testing Verification and Reliability*, v. 11, n. 2, p. 97–111, cited By (since 1996) 25, 2001.

El Ariss, O.; Xu, D.; Dandey, S.; Vender, B.; McClean, P.; Slator, B. A systematic capture and replay strategy for testing complex gui based java applications. In: *Seventh International Conference on Information Technology: New Generations (ITNG).*, 2010, p. 1038 –1043.

Engels, G.; Galdali, B.; Lohmann, M. Towards model-driven unit testing. 2007. Available on: http://dx.doi.org/10.1007/978-3-540-69489-2_23

Faitelson, D.; Tyszberowicz, S. Data refinement based testing. *International Journal of Systems Assurance Engineering and Management*, v. 2, p. 144–154, 10.1007/s13198-011-0060-y, 2011.
Available on: http://dx.doi.org/10.1007/s13198-011-0060-y

Felder, M.; Morzenti, A. Validating real-time systems by history-checking TRIO specifications. In: *Software Engineering, 1992. International Conference on*, 1992, p. 199 –211.

Fraser, G.; Zeller, A. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, v. 38, n. 2, p. 278 –292, 2012.

Gargantini, A.; Riccobene, E. Asm-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, v. 7, n. 11, p. 1050–1067, cited By (since 1996) 22, 2001.

Ghezzi, C.; Mandrioli, D.; Morzenti, A. Trio: A logic language for executable specifications of real-time systems. *J. Syst. Softw.*, v. 12, 1990.

Giannakopoulou, D.; Bushnell, D.; Schumann, J.; Erzberger, H.; Heere, K. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence*, v. 63, p. 5–30, 10.1007/s10472-011-9224-3, 2011a.
Available on: http://dx.doi.org/10.1007/s10472-011-9224-3

Giannakopoulou, D.; Rungta, N.; Feary, M. Automated test case generation for an autopilot requirement prototype. In: *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, 2011b, p. 1825 –1830.

Gibson, J.; Raffy, J.-L.; Lallet, E. Formal object-oriented development of a voting system test oracle. *Innovations in Systems and Software Engineering*, v. 7, p. 237–245, 10.1007/s11334-011-0167-y, 2011.
Available on: http://dx.doi.org/10.1007/s11334-011-0167-y

Gladisch, C.; Tyszberowicz, S.; Beckert, B.; Yehudai, A. Generating regression unit tests using a combination of verification and capture & replay. In: *Proceedings of the 4th international conference on Tests and proofs*, TAP'10, Berlin, Heidelberg: Springer-Verlag, 2010, p. 61–76 (*TAP'10*,).

Goldberg, A.; Havelund, K.; McGann, C. Runtime verification for autonomous spacecraft software. In: *Aerospace Conference, 2005 IEEE*, 2005, p. 507–516.

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. *SIGPLAN Not.*, v. 10, n. 6, p. 493–510, 1975.
Available on: http://doi.acm.org/10.1145/390016.808473

GOTLIEB, A.; BERNARD, P. A semi-empirical model of test quality in symmetric testing: Application to testing java card apis. In: *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, 2006, p. 329–336.

GRIESKAMP, W.; LEPPER, M.; SCHULTE, W.; TILLMANN, N. Testable use cases in the abstract state machine language. In: *Quality Software, 2001. Proceedings.Second Asia-Pacific Conference on*, 2001, p. 167–172.

HAGAR, J. B.; M., B. J. Using formal specifications as oracles for system-critical software. *ACM Ada Letters*, v. 6, p. 55–72, 1996.

HAJI-VALIZADEH, A.; LOPARO, K. Decentralized supervisory predicate control of discrete event dynamical systems. In: *American Control Conference, 1994*, 1994, p. 1099 – 1103 vol.1.

HAKANSSON, J.; JONSSON, B.; LUNDQVIST, O. Generating online test oracles from temporal logic specifications. *International Journal on Software Tools for Technology Transfer (STTT)*, v. 4, n. 4, p. 456–471, 2003.
Available on: http://dx.doi.org/10.1007/s10009-003-0107-8

HIERONS, R. Oracles for distributed testing. *Software Engineering, IEEE Transactions on*, v. 38, n. 3, p. 629 –641, 2012.

HOFFMAN, D.; STROOPER, P. Automated module testing in prolog. *IEEE Transactions on Software Engineering*, v. 17, n. 9, p. 934–943, 1991.

HOWDEN, W. Reliability of the path analysis testing strategy. *Software Engineering, IEEE Transactions on*, v. SE-2, n. 3, p. 208 – 215, 1976.

HU, P.; ZHANG, Z.; CHAN, W.; TSE, T. An empirical comparison between direct and indirect test result checking approaches. In: *Proceedings of the Third International Workshop on Software Quality Assurance, SOQUA 2006*, Portland, OR, 2006, p. 6–13.

HUMMEL, O.; ATKINSON, C. Automated harvesting of test oracles for reliability testing. In: *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, 2005, p. 196–202 Vol. 1.

IEEE Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, p. 1 –418, 2010.

Janjic, W.; Barth, F.; Hummel, O.; Atkinson, C. Discrepancy discovery in search-enhanced testing. In: *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, New York, NY, USA: ACM, 2011, p. 21–24 (*SUITE '11*,).
Available on: `http://doi.acm.org/10.1145/1985429.1985435`

Jia, X. Model-based formal specification directed testing of abstract data types. In: *Computer Software and Applications Conference, 1993. COMPSAC 93. Proceedings., Seventeenth Annual International*, 1993, p. 360–366.

Jin, H.; Wang, Y.; Chen, N.-W.; Gou, Z.-J.; Wang, S. Artificial neural network for automatic test oracles generation. In: *Computer Science and Software Engineering, 2008 International Conference on*, 2008, p. 727–730.

Jin, H.; Wang, Y.; Chen, N.-W.; Wang, S.; Zeng, L.-M. Predication of program behaviours for functionality testing. In: *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, Washington, DC, USA: IEEE Computer Society, 2009, p. 4993–4996 (*ICISE '09*,).

Jonsson, B.; Padilla, G. An execution semantics for msc-2000. 2001.
Available on: `http://dx.doi.org/10.1007/3-540-48213-X_23`

Jost, J. *Dynamical systems : examples of complex behaviour*. Springer, 2005.

Kanstren, T. Program comprehension for user-assisted test oracle generation. In: *ICSEA '09: Fourth International Conference on Software Engineering Advances.*, 2009, p. 118–127.

Kim-Park, D.; Riva, C.; Tuya, J. A partial test oracle for xml query testing. In: *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009. TAIC PART '09.*, 2009, p. 13 –20.

Kim-Park, D. S.; Riva, C.; Tuya, J. An automated test oracle for xml processing programs. In: *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, New York, NY, USA: ACM, 2010, p. 5–12 (*STOV '10*,).

Kuhn, D. R.; Okum, V. Pseudo-exhaustive testing for software. In: *Software Engineering Workshop, 2006. SEW '06. 30th Annual IEEE/NASA*, 2006, p. 153–158.

Kuo, F.-C.; Zhou, Z.; Ma, J.; Zhang, G. Metamorphic testing of decision support systems: a case study. *Software, IET*, v. 4, n. 4, p. 294 –301, 2010.

Lamancha, B.; Polo, M.; Caivano, D.; Piattini, M.; Visaggio, G. Automated generation of test oracles using a model-driven approach. *Information and Software Technology*, cited By (since 1996) 0; Article in Press, 2012.
Available on: http://www.scopus.com/inward/record.url?eid=2-s2.0-84867042335&partnerID=40&md5=bf954e08591fce5c57de8927ed1ea66b

Lasalle, J.; Peureux, F.; Guillet, J. Automatic test concretization to supply end-to-end mbt for automotive mechatronic systems. In: *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, New York, NY, USA: ACM, 2011, p. 16–23 (*ETSE '11*,).
Available on: http://doi.acm.org/10.1145/2002931.2002934

Lazić, L.; Velašević, D. Applying simulation and design of experiments to the embedded software testing process: Research articles. *Softw. Test. Verif. Reliab.*, v. 14, n. 4, p. 257–282, 2004.
Available on: http://dx.doi.org/10.1002/stvr.v14:4

Leavens, G. T.; Baker, A. L.; Ruby, C. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, v. 31, n. 3, p. 1–38, 2006.
Available on: http://dx.doi.org/10.1145/1127878.1127884

Li, J.; Liu, H.; Seviora, R. Constructing automated protocol testing oracles to accommodate specification nondeterminism. In: *Proceedings of the Sixth International Conference on Computer Communications and Networks.*, 1997, p. 532–537.

Li, X.; Qiu, X.; Wang, L.; Chen, X.; Zhou, Z.; Yu, L.; Zhao, J. Uml interaction model-driven runtime verification of java programs. *Software, IET*, v. 5, n. 2, p. 142 –156, 2011.

Lin, J.-C.; Ho, I. A new perspective on formal testing method for real-time software. In: *Euromicro Conference, 2000. Proceedings of the 26th*, 2000, p. 270–276 vol.2.

Lin, J.-C.; Ho, I. Generating timed test cases with oracles for real-time software. *Advances in Engineering Software*, v. 32, n. 9, p. 705–715, 2001.

Lin, J.-C.; Yeh, P.-L.; Yang, S.-C. Promoting the software design for testability towards a partial test oracle. In: *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*, 1997, p. 209–214.

Lin, Y. *A model transformation approach to automated model evolution.* Doctoral Dissertation, University of Alabama at Birmingham, Birmingham, AL, USA, adviser-Gray, Jeffrey G., 2007.

Lozano, R. C. n.; Schulte, C.; Wahlberg, L. Testing continuous double auctions with a constraint-based oracle. In: *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, Berlin, Heidelberg: Springer-Verlag, 2010, p. 613–627 (*CP'10*,).

Lu, Y.; Ye, M. Oracle model based on rbf neural networks for automated software testing. *Information Technology Journal*, v. 6, n. 3, p. 469–474, cited By (since 1996) 1, 2007.

Luckham, D.; Von Henke, F. An overview of anna, a specification language for ada. *Software, IEEE*, v. 2, n. 2, p. 9–22, 1985.

Luqi; Yang, H.; Yang, X. Constructing an automated testing oracle: an effort to produce reliable software. In: *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*, 1994, p. 228–233.

MacColl, I.; Murray, L.; Strooper, P.; Carrington, D. Specification-based class testing: a case study. In: *Proceedings of the Second International Conference on Formal Engineering Methods.*, 1998, p. 222–231.

Machado, P. D. L.; Oliveira, E. A. S.; Barbosa, P. E. S.; Rodrigues, C. L. Testing from structured algebraic specifications: The veritas case study. *Electron. Notes Theor. Comput. Sci.*, v. 130, p. 235–261, 2005.

Manolache, L.; Kourie, D. Software testing using model programs. *Software - Practice and Experience*, v. 31, n. 13, p. 1211–1236, cited By (since 1996) 2, 2001.

Mao, Y.; Boqin, F.; Li, Z.; Yao, L. Automated test oracle based on neural networks. In: *Cognitive Informatics, 2006. ICCI 2006. 5th IEEE International Conference on*, 2006a, p. 517–522.

Mao, Y.; Boqin, F.; Li, Z.; Yao, L. Neural networks based automated test oracle for software testing. 2006b.
Available on: `http://dx.doi.org/10.1007/11893295_55`

Mathworks Best practices for large-scale modeling access on sep, 2012. `http://www.mathworks.co.uk/company/events/conferences/matlab-tour/proceedings/best-practices-for-large-scale-modeling.pdf`, 2011.

MATHWORKS About model verification blocks. r2011b documentation: Signal basics. access on jun, 2012. Available at http://www.mathworks.com/help/toolbox/simulink/ug/f15-99132.html, 2012.

MATHWORKS Documentation center: Bouncing ball subsystem. r2013a documentation: Gauges blockset. access on may, 2013. Available at http://www.mathworks.com/help/gauges/examples/bouncing-ball-subsystem.html, 2013a.

MATHWORKS Simulink: simulation and model-based design. r2013a overview. access on may, 2013. Available at http://www.mathworks.com/products/simulink/, 2013b.

MATSUMOTO, L. Y. *Simulink 7.2: Guia prático.* Editora Érica, 2008.

MAYER, J.; GUDERLEI, R. An empirical study on the selection of good metamorphic relations. In: *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, 2006a, p. 475–484.

MAYER, J.; GUDERLEI, R. On random testing of image processing applications. In: *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, 2006b, p. 85–92.

MCDONALD, J.; MURRAY, L.; STROOPER, P. Translating object-z specifications to object-oriented test oracles. *Asia-Pacific Software Engineering Conference*, p. 414–423, 1997.

MCDONALD, J.; STROOPER, P. Translating object-z specifications to passive test oracles. In: *Proceedings of the Second International Conference on Formal Engineering Methods.*, 1998, p. 165–174.

MCDONALD, J.; STROOPER, P.; HOFFMAN, D. Tool support for generating passive c++ test oracles from object-z specifications. In: *Software Engineering Conference, 2003. Tenth Asia-Pacific*, 2003, p. 322–331.

MCDOWELL, C. E.; HELMBOLD, D. P. Debugging concurrent programs. *ACM Comput. Surv.*, v. 21, n. 4, p. 593–622, 1989.
Available on: `http://doi.acm.org/10.1145/76894.76897`

MEMON, A.; BANERJEE, I.; HASHMI, N.; NAGARAJAN, A. Dart: a framework for regression testing "nightly/daily builds" of gui applications. In: *ICSM 2003: Proceedings of the International Conference on Software Maintenance.*, 2003a, p. 410–419.

MEMON, A.; BANERJEE, I.; NAGARAJAN, A. What test oracle should i use for effective gui testing? In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering.*, 2003b, p. 164–173.

MEMON, A.; NAGARAJAN, A.; XIE, Q. Automating regression testing for evolving gui software. *Journal of Software Maintenance*, v. 17, n. 1, p. 27–64, 2005.

MEMON, A.; XIE, Q. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004a, p. 8–17.

MEMON, A.; XIE, Q. Using transient/persistent errors to develop automated test oracles for event-driven software. In: *Proceedings of the 19th International Conference on Automated Software Engineering*, 2004b, p. 186–195.

MEMON, A.; XIE, Q. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, v. 31, n. 10, p. 884–896, 2005.

MEMON, A. M.; POLLACK, M. E.; SOFFA, M. L. Automated test oracles for guis. In: *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA: ACM, 2000, p. 30–39.

MEYER, B.; CIUPA, I.; LEITNER, A.; LIU, L. Automatic testing of object-oriented software. 2007.
Available on: `http://dx.doi.org/10.1007/978-3-540-69507-3_9`

MILLER, T.; STROOPER, P. Supporting the software testing process through specification animation. In: *Proceedings of the First International Conference on Software Engineering and Formal Methods*, 2003, p. 14–23.

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. *Foundations of machine learning.* The MIT Press, 2012.

MOTTU, J.-M.; BAUDRY, B.; TRAON, Y. Model transformation testing: oracle issue. In: *ICSTW'08: IEEE International Conference on Software Testing Verification and Validation Workshop.*, 2008, p. 105–112.

MURPHY, C. Using runtime testing to detect defects in applications without test oracles. In: *FSEDS '08: Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*, New York, NY, USA: ACM, 2008, p. 21–24.

Murphy, C.; Raunak, M. S.; King, A.; Chen, S.; Imbriano, C.; Kaiser, G.; Lee, I.; Sokolsky, O.; Clarke, L.; Osterweil, L. On effective testing of health care simulation software. In: *Proceedings of the 3rd Workshop on Software Engineering in Health Care*, SEHC '11, New York, NY, USA: ACM, 2011, p. 40–47 (*SEHC '11*,). Available on: `http://doi.acm.org/10.1145/1987993.1988003`

Murphy, C.; Shen, K.; Kaiser, G. Automatic system testing of programs without test oracles. In: *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, New York, NY, USA: ACM, 2009a, p. 189–200.

Murphy, C.; Shen, K.; Kaiser, G. Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles. In: *ICST '09: International Conference on Software Testing Verification and Validation.*, 2009b, p. 436–445.

Myers, G. *The art of software testing.* John Wiley and Sons, 2004.

Nadeem, A.; Rehman, M. Testaf: A test automation framework for class testing using object-oriented formal specifications. *Journal of Universal Computer Science*, v. 11, n. 6, p. 962–985, cited By (since 1996) 0, 2005.

Nardi, P. A.; Delamaro, M. E. *Test oracles associated with dynamical system models.* Technical Report, Universidade de São Paulo/São Carlos - ICMC, 2011.

O'Malley, T. O. *A model of specification-based test oracles.* Doctoral Dissertation, University of California, Irvine, chair-Richardson, Debra J., 1996.

Packevičius, v.; Ušaniov, A.; Bareiša, E. Software testing using imprecise ocl constraints as oracles. In: *CompSysTech '07: Proceedings of the 2007 international conference on Computer systems and technologies*, New York, NY, USA: ACM, 2007, p. 1–6.

Pasala, A.; Rao, S.; Gupta, A.; Gunturu, S. On the validation of api execution-sequence to assess the correctness of application upon cots upgrades deployment. In: *ICCBSS '07: Sixth International IEEE Conference o Commercial-off-the-Shelf (COTS)-Based Software Systems.*, 2007, p. 225–232.

Peters, D.; Parnas, D. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, v. 24, n. 3, p. 161–173, 1998.

Peters, D.; Parnas, D. L. Generating a test oracle from program documentation: work in progress. In: *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA: ACM, 1994, p. 58–65.

Peters, D. b.; Parnas, D. c. Requirements-based monitors for real-time systems. *IEEE Transactions on Software Engineering*, v. 28, n. 2, p. 146–158, cited By (since 1996) 12, 2002.

Pilskalns, O.; Andrews, A.; Knight, A.; Ghosh, S.; France, R. Testing uml designs. *Inf. Softw. Technol.*, v. 49, n. 8, p. 892–912, 2007.

Pilskalns, O. J. *Unified modeling language design testing and analysis*. Doctoral Dissertation, Washington State University, Pullman, WA, USA, chair-Andrews, Anneliese, 2004.

Qu, G.; Guo, S.-T.; Zhang, H. A practical approach to assertion testing framework based on inner class. In: *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, 2011, p. 133 –137.

Rajan, A.; Bousquet, L.; Ledru, Y.; Vega, G.; Richier, J.-L. Assertion-based test oracles for home automation systems. In: *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, MOMPES '10, New York, NY, USA: ACM, 2010, p. 45–52 (*MOMPES '10*,).
Available on: `http://doi.acm.org/10.1145/1865875.1865882`

Reactive Systems *Reactis user's guide*. Reactive Systems, inc, 2012.

Reicherdt, R.; Glesner, S. Slicing matlab simulink models. In: *34th International Conference on Software Engineering (ICSE)*, 2012, p. 551 –561.

Richardson, D.; Aha, S.; O'Malley, T. Specification-based test oracles for reactive systems. In: *International Conference on Software Engineering.*, 1992, p. 105–118.

Richardson, D. J. Taos: Testing with analysis and oracle support. In: *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA: ACM, 1994, p. 138–153.

Sangwan, O. P.; Bhatia, P. K.; Singh, Y. Radial basis function neural network based approach to test oracle. *SIGSOFT Softw. Eng. Notes*, v. 36, n. 5, p. 1–5, 2011.
Available on: `http://doi.acm.org/10.1145/2020976.2020992`

Seifert, D. Conformance testing based on uml state machines. In: *ICFEM '08: Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, Berlin, Heidelberg: Springer-Verlag, 2008, p. 45–65.

Shahamiri, S.; Kadir, W.; Ibrahim, S.; Hashim, S. An automated framework for software test oracle. *Information and Software Technology*, v. 53, n. 7, p. 774–788,

cited By (since 1996) 3, 2011.
Available on: http://www.scopus.com/inward/record.url?eid=2-s2.0-79955055107&partnerID=40&md5=42afa6184a97f0086c450e9d916f34ab

Shahamiri, S.; Kadir, W.; Mohd-Hashim, S. A comparative study on automated software test oracle methods. In: *ICSEA '09: Fourth International Conference on Software Engineering Advances.*, 2009, p. 140–145.

Shahamiri, S.; Wan Kadir, W.; Ibrahim, S. A single-network ann-based oracle to verify logical software modules. In: *2nd International Conference on Software Technology and Engineering (ICSTE).*, 2010, p. V2–272 –V2–276.

Shahamiri, S.; Wan-Kadir, W.; Ibrahim, S.; Hashim, S. Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering*, v. 19, p. 303–334, 10.1007/s10515-011-0094-z, 2012.
Available on: http://dx.doi.org/10.1007/s10515-011-0094-z

Shailesh, S. *Ease of analysing large signal modeling.* S. E. Asia eNews, access on Sep, 2012. Available at http://techsource-asia.com/edm/2011Aug/ideas.html, 2011.

Shawcross, P. *Flightpath glossary of aviation terms.* Cambridge University Press, 2011.
Available on: http://www.cambridge.org/gb/elt/catalogue/subject/project/custom/item6604469/Flightpath-Glossary-of-Aviation-Terms/

Shimeall, T.; Leveson, N. An empirical comparison of software fault tolerance and fault elimination. In: *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, 1988, p. 180–187.

Shrestha, K.; Rutherford, M. An empirical evaluation of assertions as oracles. In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, 2011, p. 110 –119.

Shukla, R.; Carrington, D.; Strooper, P. A passive test oracle using a component's api. In: *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, 2005, p. 7 pp.–.

Silva, J.; Campos, J.; Paiva, A. Model-based user interface testing with spec explorer and concurtasktrees. *Electronic Notes in Theoretical Computer Science*, v. 208, p. 77–93, 2008.

Skroch, O. Validation of component-based software with a customer centric domain level approach. In: *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, 2007, p. 459–466.

Stewart, I. *Does god play dice: The new mathematics of chaos.* Blackwell Publishing, 1989.

Stocks, P.; Carrington, D. Test templates: a specification-based testing framework. In: *Software Engineering, 1993. Proceedings., 15th International Conference on*, 1993, p. 405–414.

Stocks, P.; Carrington, D. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, v. 22, n. 11, p. 777–793, 1996.

Sun, C.; Wang, G.; Mu, B.; Liu, H.; Wang, Z.; Chen, T. Metamorphic testing for web services: Framework and a case study. In: *Web Services (ICWS), 2011 IEEE International Conference on*, 2011, p. 283 –290.

Svendsen, A.; Haugen, O.; Møller-Pedersen, B. Specifying a testing oracle for train stations. In: *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, New York, NY, USA: ACM, 2011, p. 5:1–5:6 (*MoDeVVa*,).
Available on: `http://doi.acm.org/10.1145/2095654.2095661`

Taneja, K.; Li, N.; Marri, M. R.; Xie, T.; Tillmann, N. Mitv: multiple-implementation testing of user-input validators for web applications. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, New York, NY, USA: ACM, 2010, p. 131–134 (*ASE '10*,).

Tiwari, S.; Mishra, K.; Kumar, A.; Misra, A. Spectrum-based fault localization in regression testing. In: *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, 2011, p. 191 –195.

Tsai, W.-T.; Chen, Y.; Paul, R.; Huang, H.; Zhou, X.; Wei, X. Adaptive testing, oracle generation, and test case ranking for web services. In: *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, 2005a, p. 101–106 Vol. 2.

Tsai, W.-T.; Chen, Y.; Zhang, D.; Huang, H. Voting multi-dimensional data with deviations for web services under group testing. In: *25th IEEE International Conference on Distributed Computing Systems Workshops.*, 2005b, p. 65–71.

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. SCALCOM-EMBEDDEDCOM'09.*, 2009, p. 351–354.

Vanmali, M.; Last, M.; Kandel, A. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, v. 17, n. 1, p. 45–62, cited By (since 1996) 8, 2002.

Vapnik, V. N. *The nature of statistical learning theory.* New York, NY, USA: Springer-Verlag New York, Inc., 1995.

Wang, F.; Yao, L.-W.; Wu, J.-H. Intelligent test oracle construction for reactive systems without explicit specifications. In: *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, p. 89 –96.

Wang, X.; Yan, Q.; Mao, X.; Qi, Z. Generating test oracle for role binding in multi-agent systems. In: *Software Engineering Conference, 2003. Tenth Asia-Pacific*, 2003, p. 108–114.

Wang, X.; Zhi-Chang; Li, Q. S. An optimized method for automatic test oracle generation from real-time specification. In: *ICECCS 2005: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems.*, 2005, p. 440–449.

Weyuker, E. J. On testing non-testable programs. *The Computer Journal*, v. 25, n. 4, p. 465–470, 1982.

Wohlin, C.; Runeson, P.; Host, M.; Ohlsson, C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering: an Introduction.* Kluver Academic Publishers, 2000.

Xie, Q. *Developing cost-effective model-based techniques for gui testing.* Doctoral Dissertation, University of Maryland at College Park, College Park, MD, USA, adviser-Memon, Atif, 2006a.

Xie, Q.; Memon, A. M. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, v. 16, n. 1, p. 4, 2007.

Xie, T. Augmenting automatically generated unit-test suites with regression oracle checking. 2006b.
Available on: `http://dx.doi.org/10.1007/11785477_23`

Xie, X.; Ho, J.; Murphy, C.; Kaiser, G.; Xu, B.; Chen, T. Y. Application of metamorphic testing to supervised classifiers. In: *QSIC '09: 9th International Conference on Quality Software.*, 2009, p. 135–144.

Xie, X.; Ho, J. W. K.; Murphy, C.; Kaiser, G.; Xu, B.; Chen, T. Y. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, article in Press, 2010.

Xie, X.; Wong, W.; Chen, T.; Xu, B. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, cited By (since 1996) 0; Article in Press, 2012.
Available on: `http://www.scopus.com/inward/record.url?eid=2-s2.0-84866085646&partnerID=40&md5=a01ccf36a2b9efe4bb04b6ac1d33a962`

Xie, X.; Wong, W.; Chen, T. Y.; Xu, B. Spectrum-based fault localization: Testing oracles are no longer mandatory. In: *Quality Software (QSIC), 2011 11th International Conference on*, 2011, p. 1 –10.

Xing, X.; Jiang, F. Gui test case definition with ttcn-3. In: *CiSE 2009: International Conference on Computational Intelligence and Software Engineering.*, 2009, p. 1–5.

Yan, C. H. The application of an algebraic design method to deal with oracle problem in object-oriented class level testing. In: *IEEE SMC '99: Conference Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics.*, 1999, p. 928–932 vol.1.

Yang, R.; Chen, Z.; Xu, B.; Wong, W.; Zhang, J. Improve the effectiveness of test case generation on efsm via automatic path feasibility analysis. In: *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, 2011, p. 17 –24.

Ye, M.; Feng, B.; Zhu, L.; Lin, Y. Automated test oracle based on neural networks. In: *ICCI 2006: 5th IEEE International Conference on Cognitive Informatics.*, 2006, p. 517–522.

Yoo, S. Metamorphic testing of stochastic optimisation. In: *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW).*, 2010, p. 192 –201.

Zeil, S. Perturbation techniques for detecting domain errors. *Software Engineering, IEEE Transactions on*, v. 15, n. 6, p. 737–746, 1989.

Zhang, J.; Yang, R.; Chen, Z.; Zhao, Z.; Xu, B. Automated efsm-based test case generation with scatter search. In: *Automation of Software Test (AST), 2012 7th International Workshop on*, 2012, p. 76 –82.

Zhang, Z.-Y.; Chan, W.; Tse, T.; Hu, P. Experimental study to compare the use of metamorphic testing and assertion checking. *Ruan Jian Xue Bao/Journal of Software*, v. 20, n. 10, p. 2637–2654, cited By (since 1996) 0, 2009.

Zheng, W.; Ma, H.; Lyu, M.; Xie, T.; King, I. Mining test oracles of web search engines. In: *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, p. 408 –411.

Zhou, L.; Ping, J.; Xiao, H.; Wang, Z.; Pu, G.; Ding, Z. Automatically testing web services choreography with assertions. In: Dong, J.; Zhu, H., eds. *Formal Methods and Software Engineering*, Springer Berlin, 2010, p. 138–154 (*Lecture Notes in Computer Science*, v.6447).

Zhu, H. A note on test oracles and semantics of algebraic specifications. In: *Quality Software, 2003. Proceedings. Third International Conference on*, 2003, p. 91–98.

Appendix

A

Aviation Glossary

The aviation terms here presented were extracted from Flightpath Glossary of Aviation Terms [1] (Shawcross, 2011).

Angle of attack	The angle between the chord line of the wing of an aircraft and the vector representing the relative motion between the aircraft and the atmosphere. Information from the angle of attack sensor, or alpha probe, is used to trigger a stall warning.
Pitch angle	the acute angle between the longitudinal axis of an aircraft or spacecraft and the direction of the wind relative to the vehicle.
Pitch axis	axis perpendicular to the yaw axis and is parallel to the plane of the wings with its origin at the center of gravity and directed towards the right wing tip.
Pitch motion	an up or down movement of the nose of the aircraft w.r.t. the pitch axis.
Roll angle	the acute angle between the roll axis of an aircraft or spacecraft and a horizontal plane.

[1] Available on: http://www.cambridge.org/gb/elt/catalogue/subject/project/custom/item6604469/Flightpath-Glossary-of-Aviation-Terms/

Roll axis	axis perpendicular to the other pitch and yaw axes with its origin at the center of gravity, and is directed towards the nose of the aircraft.
Rolling motion	an up and down movement of the wing tips of the aircraft, w.r.t. the roll axis.
Stall	a condition of an aircraft in flight in which a reduction in speed or an increase in the aircraft's angle of attack causes a sudden loss of lift resulting in a downward plunge.
Waypoint	a point on the journey to the final destination.
Yaw angle	the acute angle between the yaw axis of an aircraft or spacecraft and a given reference direction, as viewed from above.
Yaw axis	axis perpendicular to the plane of the wings with its origin at the center of gravity and directed towards the bottom of the aircraft.
Yaw motion	a movement of the nose of the aircraft from side to side, w.r.t. the yaw axis.