Test oracles associated with dynamical systems models

Paulo Augusto Nardi

Márcio Eduardo Delamaro

Abstract

Dynamical systems are present in many fields of application where failure can cause deaths, high economic losses or damage to the environment. A characteristic of such systems is the large amount of input and output data, making the test phase particularly challenging without the use of automation. There are tools that support the development of models for analysis and simulation before the implementation. But if such models are not in agreement with the specification, analysis and simulation may be inaccurate with regard to the implementation.

The objective of this technical report is presenting a systematic review on the "test oracles associated with dynamical systems models" theme, to identify works that address oracles applied to such systems, as well as major limitations related to their applications and oracles support tools.

Contents

List of Figures

List of Tables

Chapter

1

Introduction

A dynamical system consists of a set of possible states, together with a rule that determines the present state in terms of past states (Alligood et al., 2000). In the context of programming, the outputs depend not only on inputs at the present instant but on passed times and the states previously assumed by the system. Examples of dynamical systems are found in aviation, as altimeters, landing gear regulators, air traffic controllers, and several other critical areas.

There are support tools such as Simulink and Scicos, which allow the modeling, simulation and analysis of dynamical systems. Models developed on these tools can be implemented and tested. However, the execution of a model may require a large amount of input data even for short periods of time, making testing process automation particularly challenging.

According to Chen et al. (2001), software testing has two limitations: the reliable test set problem and the oracle problem. The former stems from the statement that a set of reliable test is one that implies the correctness of the program and, consequently, a finite set of reliable test is not attainable in general. The second problem is to decide whether the obtained result is equal to the expected result. This role is played by the oracle, which often is the tester herself/himself.

The TeTooDS (Araujo e Delamaro, 2008) is an example of tool that focuses on the reliable test set problem. This tool supports the testing process by automating the input data selection and it is integrated with Simulink as follows: the TeTooDS generates the

input data, it runs the model by calling the Simulink and captures the generated output data.

The decision about the output correctness is crucial in the automation of testing activity for dynamical systems. As an illustration, one can imagine a system that receives 500 inputs every 100 milliseconds. So to test the behavior of this system for one second, it would require 5000 inputs. One hour of simulation would require 18 million of inputs (Araujo, 2008). Assuming that at least the same amount of outputs is produced, a manual comparison between expected and obtained results would become impracticable.

The oracle must have prior knowledge about what should be the expected output for a given input. But comparing the expected output and the obtained output is not always possible or feasible. A program is considered non-testable if there is not an oracle, or if it is theoretically possible but very difficult in practice, to determine the correct output (Weyuker, 1982). Programs that produce a large amount of outputs so that is impractical to check them are identified as non-testable programs, such as dynamical systems.

Although, for non-testable programs, there are no oracles to identify the correct result for all pairs of input and output, it is possible to build partial oracles that determine whether the outputs are in compliance with certain requirements or expected characteristics. The knowledge used by the oracle in such cases can be based on informal specifications, stored results of the execution of other programs (Smeulders et al., 2000) or based on formal models (Hagar e Bieman, 1996). The oracles based on informal specifications are difficult to be automated, since the decision about the correctness of a particular execution depends on the interpretation of the specification. Oracles based on stored results can be applied when there is a database of previously stored cases, such as records of diagnostic imaging. Such images are compared with a current image to define the diagnosis. Oracles based on stored results can be used in regression testing, where results from previous and stable versions are compared with the current version in testing.

Given the possibly critical nature of the dynamical systems and the fact that these are often embedded, it is clear the motivation for a study on the topic “oracles associated with dynamical systems models”. This systematic review aims to (i) identify the test oracles applicable to dynamical systems, (ii) to identify the supporting tools for test oracles and (iii) possible limitations of oracles appliance.

In Chapter 2, we present the basic concepts of software testing, test oracles, dynamical systems and systematic review. In Chapter 3, we present a systematic review on the topic “oracles associated with dynamical systems models”. In Chapter 4, we present the final remarks.

Chapter

2

Basic Concepts

This chapter presents the basic concepts of software testing, test oracles, dynamical systems and systematic review for the best understanding on the subject of this technical report.

2.1 Software Testing

Test is the process of executing a program with the goal of finding errors or failures and aims to increase confidence that the software perform the desired operations, given the restrictions imposed on it (Myers, 2004).

Recurrent software testing concepts are input domain, obtained output, expected output, test case and test set. The **input domain** is the set of all input data that can be applied to the program, also called test data. **Obtained output** is the result of the program execution and **expected output** is the one that should be produced by executing the program (or part of it) according to a given input data. A **test case** is a pair formed by the input data and expected output. **Test set** represents all the test cases used during the software testing.

According to The Institute of Electrical and Eletronics Engineers (1990), **fault** in a program is an incorrect step, procedure or definition of data. **Error** is the difference between a obtained value and the expected value. **Failure** is a notable event in which the program violates its specification. There may be situations where an error does not imply

a failure. For example, let's suppose an error occurred when a program is computing the result of a function. If this error, when propagated through the system, does not influence the final result, visible to the user, there is an error but not a failure.

The tests applied to procedures and functions, individually, are called **unit testing** (Kaner et al., 1999). The test of the combination between the units is called **integration testing**. The test applied to the whole system is the **system testing**.

When a program is tested, it is convenient to create a mechanism that allows the tester to determine which input data will be used so that the same set of test cases can be executed several times. With that mechanism, the tester does not need to enter the same values repeatedly to each execution. Also, when one is performing a unit testing, it may be appropriated to isolate and execute just the procedure or the function to be tested. Such mechanisms are called **drivers**. According to Ammann e Offutt (2008), a driver is a software component or test tool that replaces the component that controls and/or calls another component (the part to be tested).

2.2 Test Oracle

Oracle is a mechanism that determines whether a system has passed or failed a test (Tu et al., 2009). This function is exercised by the tester, or by automated means or a hybrid. According to Shahamiri et al. (2009) the possible assignments of an oracle are: generate the expected outputs, storing them, comparing the expected and obtained outputs, and decide if there is a failure or not.

An ideal automated oracle should have the equivalent behavior to the application under test, but completely reliable. It should accept all entries for the specified application and always produce the correct result (Mao et al., 2006a). Furthermore, a test oracle just need to have the answers to the data that is actually used in the test.

However, there are programs in which the identification of the results is impossible or impractical, called non-testable programs (Weyuker, 1982). The difficulty in interpreting test results is known as the oracle problem (claude Gaudel, 1995). Given that a tester can make a mistake while calculating an expected output and the large number of outputs to be compared during the test phase illustrate the obvious interest in creating automated oracles. There are works that address the automation of oracles for non-testable programs in order to partially solve the oracle problem.

Two approaches are the partial oracles or pseudo-oracles. According to Davis e Weyuker (1981), **pseudo-oracles** are programs (executable models or code) written in parallel to a system development, by a second team, following the same specifications. The two programs, the oracle and the software under test, run with the same input data and outputs

are compared. If the outputs are equal or are within an acceptable margin of accuracy, it is considered that the original program passed the test. Importantly, there is no guarantee that the oracle is free of faults. Thus, when a program obtained output is not equivalent to the oracle obtained output, one must go through a debug process to check which of the two actually has the fault.

Partial oracles are able to identify if the result is incorrect, even without knowledge of the correct output (Weyuker, 1982). The verification is based on specifications, written as constraints such as contracts (pre and post-conditions) and invariant (Kim-Park et al., 2009). Pre-conditions, post-conditions and invariants are expressions that must be satisfied, respectively, before, during and after an execution. As an example, a partial oracle for program that calculates the sine function can be based on the post-condition which describes that the result should be contained in the interval between -1 and 1. Any result outside this range should be reported as an error.

According to Peters e Parnas (2002), a false negative result occurs when the oracle reports that an acceptable behavior is unacceptable. A false positive occurs when the oracle reports that an unacceptable behavior is acceptable. Thus, the partial oracle described above will not cause false negative results, because the sine function, in fact, does not generate values less than -1 or greater than 1. But it can cause a false positive, i.e., if the program states that $sin(90) = 0.5$, the oracle will provide a "pass" result, according to the given post-condition, but the sin of 90 degrees is 1.

A test oracle should have knowledge about the expected output and make the comparison with the obtained output, so it is composed of oracle information and oracle procedure (Memon et al., 2003b). **Oracle information** represents the expected output. The information is obtained through the following sources: specification, previously stored results, from the execution of a code developed in parallel, metamorphic relations (discussed in the next chapter) or neural networks. Information can be a concrete result (a value), or a higher level of abstraction information, as the sine function example using the ranges between -1 and 1. An **oracle procedure** compares the oracle information with the obtained output. This comparison is performed at runtime (**online**), or after the execution (**offline**) (Durrieu et al., 2009).

2.3 Dynamical Systems

A dynamical system consists of a set of possible states, together with a rule that determines the present state in terms of previous states (Alligood et al., 2000). According to Jost (2005), it is a system that evolves over time through iterative application of a underlying

dynamical rule. This transition rule describes the change of the current status in terms of himself, and possibly also previous states.

Dynamical systems are present in several application areas as physics, economics and biology. Examples are climate models which represent the interaction between ocean and atmosphere (Neelin et al., 1994). In such cases, accurate simulations of such models are important for predicting climate change, with implications in the area of cultivation, cyclones warnings, in addition to acquiring greater basis for distinguishing the plausible speculation regarding global climate change.

In economics, Adam Smith observed a free market system, established by the Law of Supply and Demand. An environment of freedom governed by the "state of law" should be stable. There is, in this system, two types of prices: the market governed by supply and demand, and the natural established by production costs (Geromel e Palhares, 2004). The bad prediction on the market equilibrium, for example, can cause economic losses of global consequences (Aghion et al., 2004), such as the Economic Crisis of 2008 and 2009. Dynamical systems are also applied in biology, as in the study of long-term behavior of population growth models in variable environments (Zhao, 2003).

Such systems can be **discrete** or **continuous**. Discrete systems, also called discrete-time systems, are those in which time is expressed by natural numbers (Maler, 1998) and there is no point in measuring the time in non-integer values. Continuous systems, or continuous time systems, are those in which time is expressed by real numbers (Maler, 1998). Dynamical systems are called **stochastic** when there is some form of uncertainty such that, from a state and the transition rule, there is more than one possible resulting state. This uncertainty can be specified mathematically by probability (Söderström, 2002). In **deterministic** systems, the change from one state implies only one possible next state.

2.4 Systematic Review

According to Kitchenham (2004), systematic review is a way to identify, evaluate and interpret the relevant research available to a particular research question, topic or phenomenon of interest. It can be divided into three phases (Biolchini et al., 2007): planning, execution and analysis result.

In the planning phase, the research goals and research questions are identified (and must be answered until the end of the review). The method to be applied in the execution is registered, such as which engines will be used to search the articles, the research question, the search string, inclusion and exclusion criteria that will guide the article selections, languages to be considered, keywords, and quality criteria to guide the reviewer to interpret the results.

In the execution phase, the objective is the collection and analysis of primary studies, i.e., publications and other sources of study related to the research question.

In the analysis result phase, the results of the primary studies that meet the purpose of the review are extracted and synthesized.

Among the reasons for conducting systematic reviews, Kitchenham (2004) cites the summarization of existing evidence concerning some technology and identifying the lack of studies in a particular line of research aiming to be suggested as future research areas.

2.5 Final Remarks

The objective of this work is a systematic review about "oracles associated with dynamical systems models". In this chapter, we presented the basic concepts for the understanding about the theme, divided in four sections: software testing, test oracles, dynamical systems and systematic review.

Chapter

3

Systematic Review

This chapter presents details related to the preparation and conduct of a systematic review (Kitchenham, 2004), in which the goal is to identify papers that have one or more items of interest related to the theme "oracles associated with models for dynamical systems". Among the existing procedures, it was used the suggested by Biolchini et al. (2007).

In addition to identifying papers that address oracles applied to dynamical systems, the review aimed to identify the main limitations related to their applications and tools that support their use.

3.1 Planning

The planning of the systematic review was conducted in accordance with the standard protocol presented by Biolchini et al. (2007). This section presents the main topics of the planning.

3.1.1 Research Objectives

This subsection elucidates the research objectives. They are used to formulate the questions to be answered by the systematic review.

- **Objective 1:** identifying types of oracles applicable to dynamical systems;

- **Objective 2:** identifying major obstacles involving the use of automated test oracles applicable to dynamical systems;
- **Objective 3:** identifying tools that support test oracles for dynamical systems.

3.1.2 Research Question Formulation

The research questions are used to identify and delineate the activity scope of the systematic review.

- *Research Questions:*
  - *1.* What kinds of oracles are applicable to dynamical systems?
  - *2.* What are the limitations observed in using these oracles?
  - *3.* There are tools that support oracles for dynamical systems?

For the best understanding about the scope and specificity of the questions, we describe the population, intervention, control, performance and application. Population corresponds to the group being observed in the study. Intervention indicates what should be observed in the research, in a given population. Control is the set of initial data that the researcher already has. Outcomes represent what is expected to reach at the end of the systematic review. Application indicates what kind of professional and areas will benefit from the systematic review outcomes.

They are:

- *Population:* research about software testing.
- *Intervention:* test oracles.
- *Control:* collection of articles, dissertations related to testing and test oracles for embedded systems.
- *Outcomes:* oracles applicable to dynamical systems (question 1); limitations of using oracles applicable to dynamical systems (question 2); list of tools that support oracles for dynamical systems (question 3).
- *Aplication:* researchers and developers involved with the development of dynamical systems.

3.1.3 Search Strategy for the Selection of Primary Studies

The search strategy and selection of primary studies were defined according to the sources of studies, keywords, and formulation of research questions:

- *Source definition criteria:* article availability via web.
- *Source list:* IEEE, ACM-Digital Library, Spriger, Scirus, Scopus, specialists.
- *keywords:* test oracle, automated oracle, dynamic systems, dynamical systems, embedded systems, Scicos, Simulink, critical systems.
- *Kinds of primary studies:* articles contemplating qualitative analysis, solution proposal and/or experimental.
- *Primary studies idiom:* English and Portuguese.

3.1.4 Criteria and Procedure for Selection of Studies

The inclusion and exclusion criteria are applied to identify the primary studies that come from direct evidence about the research questions. The quality criteria aims to cataloging the items according to the objectives of the systematic review.

Inclusion Criteria

Except for those that meet at least one exclusion criterion, it should be included in the review the papers that:

- *Inclusion Criterion 1:* describe how a test oracle can be generated automatically;
- *Inclusion Criterion 2:* identify a test oracle and its definition or application;
- *Inclusion Criterion 3:* identify tools that support oracles;
- *Inclusion Criterion 4:* addresses the limitations of the oracle uses.

Exclusion Criteria

Even attending at least one inclusion criterion, should be excluded the papers that:

- *Exclusion Criterion 1:* encompass oracles not applicable to dynamical systems;
- *Exclusion Criterion 2:* do not present definition, classification, limitations, tools or description about automatic oracle generation.

Quality Criteria

The following quality criteria were defined:

- *Quality Criterion 1:* the paper mentions the oracle in the context of embedded, dynamical, real-time or critical systems;
- *Quality Criterion 2:* the paper mentions oracles in the context of Simulink or Scicos;
- *Quality Criterion 3:* the paper presents comparative analysis between different categories of oracles;
- *Quality Criterion 4:* the paper mentions oracle categories;
- *Quality Criterion 5:* the paper describes oracle limitations.

3.1.5 Selection Process of the Primary Studies

The selection process of the primary studies was divided into preliminary and final, as presented in the next subsections.

Preliminary selection process

The preliminary selection process consisted of the following steps:

1. Search, in each source, for articles based on the defined string;
2. Referral to specialists;
3. Selection of articles by reading the abstract and introduction, considering the inclusion and exclusion criteria. A search for the term "oracle" was accomplished on the body of the articles so that the second exclusion criterion could be observed;
4. Cataloging of the selected and unselected articles with their justifications.

Final selection process

The selected articles were reviewed by complete reading. Those which contents do not fit the inclusion criteria were separated and placed together with non-selected works with their respective justifications.

Evaluation of the primary studies quality

The evaluation of the primary studies quality is based on the previously established quality criteria and aims to catalog the items according to the systematic review objectives. This assessment is not intended to create a hierarchy of relevance of the presented papers.

The articles were classified into the following categories:

1. Oracles in the context of embedded, dynamic, real-time or critical systems;
2. Test oracles for Simulink or Scicos;
3. Oracle generation;
4. Classification and comparison between test oracles;
5. Test oracle limitations.

3.1.6 Strategies for Extraction and Summarization of Results

The summary of the results is divided into subsections according to the objectives of this systematic review and the considered categories of test oracles, therefore the same article may be referenced in different categories. The grouping of summarization aims to facilitate the visualization of the review outcomes.

3.2 Conduction

The systematic review was conducted during a period of four months (January/2010 to April/2010 and updated on January/2011). After applying the search string to the respective engines, 493 works were returned. They were submitted to the stages of preliminary selection, in which 125 remained at the end of the final selection process. The next sections present more details of the activities, including the strategy adopted for building the search strings and results.

3.2.1 Preliminary Selection

The preliminary selection was conducted in three steps: construction of search string, searches on the six cited sources (Subsection 3.1.3) and paper elimination. The subsequent sections describe these steps.

- **Search string construction**

For the search string construction, we initially considered the following keywords, based on the research questions and the items related to the scope and specifics (both in Subsection 3.1.2):

test oracle, automated oracle, dynamic systems, dynamical systems, embedded systems, Scicos, Simulink, critical systems

Given the low amount of work returned by the original string, we expanded the search, simplifying the string as follow:

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

- **Search and Paper Elimination**

At the end of search on the five electronic sources (IEEE, ACM, Springer, Scirus and Scopus), we obtained 493 results with duplicate articles in the same and different sources.

During the pre-selection, we eliminated: duplicated results, results that did not have any keywords in the abstract or title; unavailable articles and those written in another language that not the Portuguese or English. In the process of elimination, all the abstracts were read.

Table 3.1 presents the relationship between pre-selected and discarded results.

Table 3.1: Pre-selected and Discarded Paper Relation

Source	**Selected**	**%**	**Non-Selected**	**%**	**Total**
IEEE	103	100,00	0	0,00	103
ACM	63	42,00	87	58,00	150
Springer	43	97,73	1	2,27	44
Scirus	34	91,89	3	8,11	37
Scopus	69	43,40	90	56,60	159
Total	312	63,29	181	36,71	493

Table 3.2 presents the numbers of duplicated articles in their respective search engines. The columns represent the sources from which the items were duplicates. For example, the IEEE did not return articles from other sources. In ACM, from the 150 returned items, 66 were duplicates from IEEE, 4 from ACM's itself, 17 from Springer, none from Scirus and Scopus, totaling 58.00% of duplicated articles.

Table 3.2: Duplicated Papers

		Duplicated					
	Returned	**IEEE**	**ACM**	**Springer**	**Scirus**	**Scopus**	**Duplicated**
IEEE	103	0	0	0	0	0	0%
ACM	150	66	4	17	0	0	58,00%
Springer	44	1	0	0	0	0	2,27%
Scirus	37	1	2	0	0	0	8,11%
Scopus	159	42	28	14	6	0	56,60%

3.3 Final Selection Result Extraction

This section presents the obtained results by conducting a systematic review. In the final selection, we fully read all pre-selected papers and discarded the ones that were in line with the exclusion criteria or did not met any inclusion criteria. A total of 125 papers were selected.

Table 3.3 presents the relation between selected and discarded results.

Table 3.3: Selected and discarded Papers Relation

Fonte	**Selected**	**%**	**Non-Selected**	**%**	**Total**
IEEE	66	64,08	37	35,92	103
ACM	28	44,44	35	55,56	63
Springer	16	37,21	27	62,79	43
Scirus	2	5,88	32	94,12	34
Scopus	13	18,84	56	81,16	69
Total	125	40,06	187	59,94	312

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, 57 articles were published, representing 45,6% of the total published in 22 years.

Figure 3.1: Publication by Year

The next subsections presents the results by question.

3.3.1 What types of oracles are applicable to dynamical systems?

From the selected articles, none had focused on the automatic generation of oracles for dynamical systems. Thus, we considered all the oracles that could, at some level, be applied to such systems. We identified four categories of oracles: specification-based, metamorphic relations, n-version and neural network.

Table 3.4: Relation between Oracle Categories

Category	Articles	%
Specification-based	90	72,00
Metamorphic relation	19	15,20
N-version or similar	11	8,80
Neural network	10	8,00

Table 3.4 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 125 or 100%. Figure 3.2 presents the relationship between the number of articles and the year in which they were published, by oracle category.

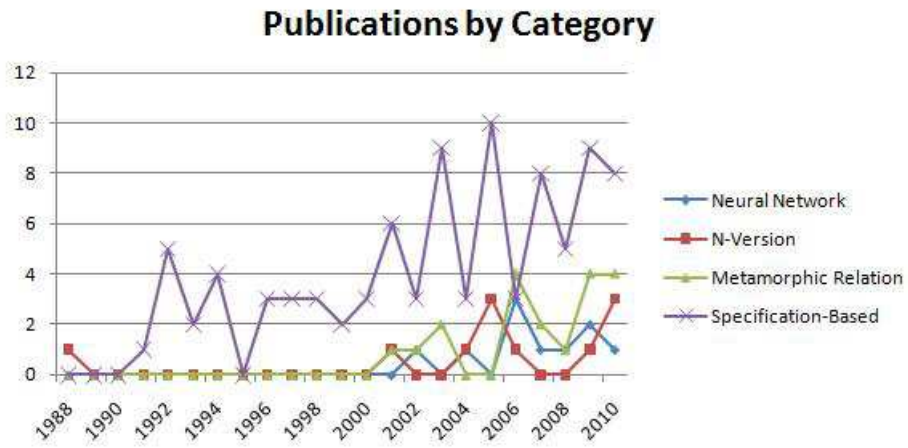


Figure 3.2: Publication by Year, by Category

Considering the inclusion and exclusion 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 33, compared to 90 published in the previous 22 years.

Studies of oracles based on metamorphic relations began in 2001 with highest number of publications in 2006, 2009 and 2010. From the 19 published articles, four have the same main author (Chen), from 2001 to 2003. This same researcher has co-authored articles in 2009 and 2010, and four other articles have, as authors, researchers who participated

as co-authors of Chen. The same network of researchers was responsible for 17 of the 19 published articles.

In relation to oracles based on neural networks, the production has increased, mainly after in 2006. Of the ten papers, four deal with continuous functions, 3 using the algorithm of backpropagation and 1 using RBF (Radial Basis Function). 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.

3.3.1.1 Specification-Based Oracles

This subsection presents an introduction to specification-based oracles and a resume about the collected data from the papers of such oracles, including a brief description about the different languages and approaches we found, as the temporal relation between them. There were papers that referenced more than one specification.

General Concepts

The formal specification of a system provides a source of information about the correct behavior of the implementation and thus it is a valuable source for test oracles (Baharom e Shukur, 2009). The specification can be used to describe the expected behavior of a system at different abstraction levels (Chen e Subramaniam, 2002). Examples of specification languages are: Notation Z, Object Z, OCL, Eiffel, VDM, JML, state machines, SDL and Mitl.

To the oracle procedure be able to check consistency between an implementation and an formal specification, implementation states are mapped to the specified objects (Aichernig, 1999) as follow: an implementation is executed with the input, called the concrete input data. A function maps the concrete input data to an input data at the same level of abstraction of the data described in the specification (called abstract data). Similarly, the concrete output data is mapped into abstract output data. A pre-condition checker evaluates the abstract input in relation to the specified and the post-conditions verifier evaluates the abstract output in relation to the specified output, considering the validated abstract input. If abstract inputs and outputs comply with the specified conditions, the test has passed (Figure 3.3, adapted from Aichernig (1999)). The function that maps concrete data into abstract data is called retrieve function (r, in Figure 3.3).

There are several approaches to mapping. There are specification languages such as Z Notation, Object Z and algebraic specifications, which allow the representation of classes.

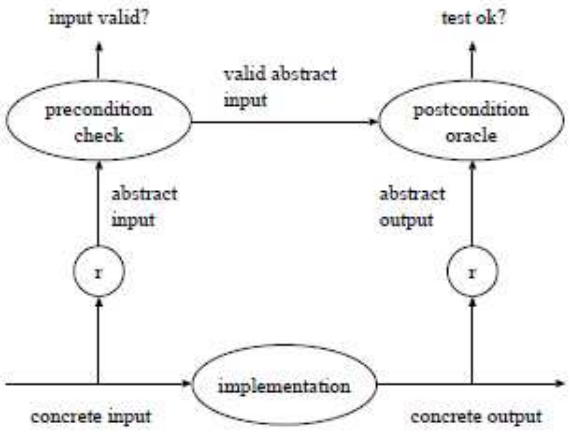


Figure 3.3: Pre-condition and post-condition based oracle.

These classes are referred as *abstract data types* (Guttag, 1977). For these languages, to one be able to create an automated oracle, there is a need for an interpreter or a compiler to translate the specification, besides the one applied to the implementation.

Another approach uses assertions, expressions that may represent invariants, pre and post-conditions inside the implementation (known as **embedded assertions**) (Baresi e Young, 2001). There are programming languages such as Java, which support assertions inserted in the code. Figure 3.4 shows an example of embedded assertion. Line 5 calls a function that returns the double of a value stored in variable. Let's suppose that the pre-condition to the operation is that the initial value must be more then or equal to 5. Line 4 contains an embedded assertion which represents that constraint. If the value is less than 5, an assertion error is raised.

```
1  .
2  .
3  .
4  assert value >= 5;
5  float doubled_value = doubleV(value);
```

Figure 3.4: Java Code With Assertion.

In the approach with the use of **wrappers**, the verification of a class does not incur in a code modification like embedded assertions. Given a class or component under test, the tester creates a second class with the same interface as the original, but with methods containing contracts to be checked. A *test driver* communicates with the wrapper that checks if the class under test complies with the specified (Figure 3.5, adapted from Edwards (2001)). The representation layer of the wrapper is responsible for the conversion of concrete values into abstract values and the abstraction layer compares the abstract values with the post-conditions. The wrapper class overwrites the public methods of

the original class. These overwritten methods call the original methods and the test is performed by running the wrapper.

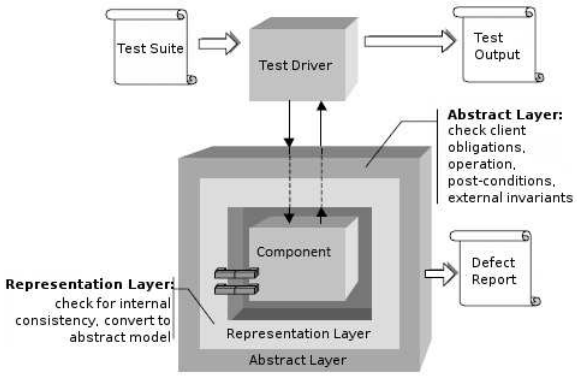


Figure 3.5: Wrapper Example

Shukla et al. (2005) gives an example with a class that represents a set of integers, containing an *insert*() method. A wrapper class, that inherits from the original class under test, contains a method with the same name. The wrapper has two other methods: one that identifies the number of elements in the set and another that compares the amount of items before and after insertion. When performing the test, instead of calling the original class, the tester calls the wrapper. The overriding method firstly identifies the amount of values in the set. Then, it calls the *insert*() method of the original class and, after the insertion operation, identifies again the number of values in the set. The wrapper compares the amount of values before and after the insertion. The amount of integers in the set after the insertion should be equals to the amount of integer before the insertion plus one. Otherwise, the insertion was not successful.

Cheon e Avila (2010) propose the use of OCL constraints translated to AspectJ as runtime oracles. These constraints are separated from the implementations code.

Collected data about specification-based oracles

Table 3.5 presents a list of different approaches to specification-based oracles and their numbers of articles. Results that have less than three publications which do not have relevant characteristics to dynamical systems, as temporal properties, were grouped in the category of "other specifications" during the process of data compilation.

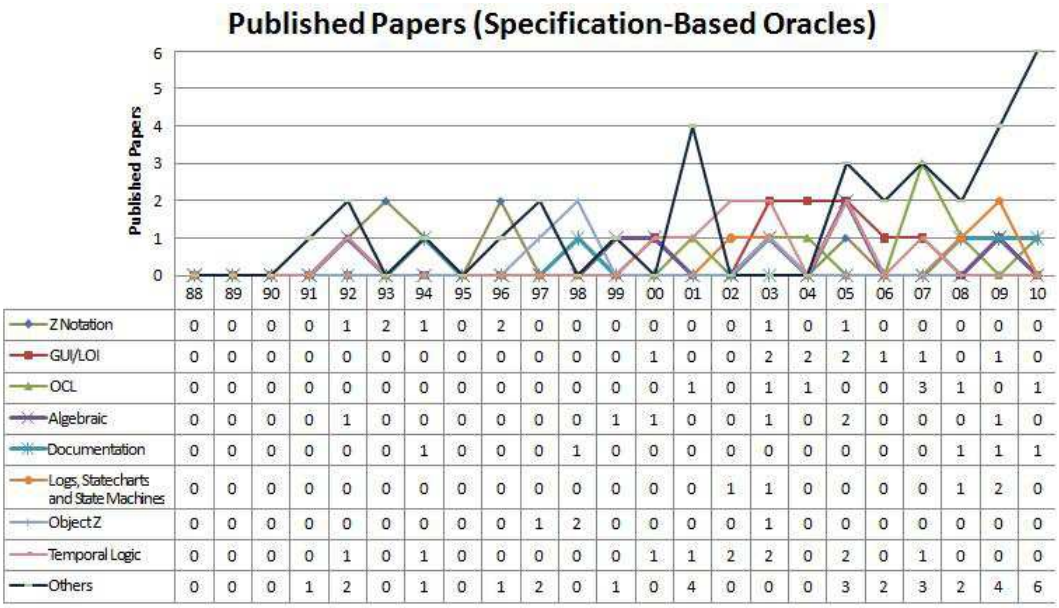
GUI/LOI is not a specification, but a classification of categories about oracle information and oracle procedure in relation to levels of detail in the construction of oracles for testing GUI applications (Graphical User Interface). We allowed their inclusion in this systematic review because we believe that the specification format can be easily adapted

Table 3.5: Specification-Based Oracles

GUI/LOI	10	OCL	8	Z Notation	8
Algebraic Specifications	7	Documentation	5	Logs and statecharts	5
Object Z	4	Temporal Logic	11	Model Checking	2
Model Transformation	3	JML	4	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	1
RSL/RAISE	1	SDL	1	TTCN-3	1
Prosper	1	WS-CDL	1	Esterel	1

to represent any kind of constraints. The number of articles 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 3.6 shows that different specifications (Others) which have not passed two publications have been proposed for use in oracles, since 1991, notably in the last years.



	88	89	90	91	92	93	94	95	96	97	98	99	00	01	02	03	04	05	06	07	08	09	10
Z Notation	0	0	0	0	1	2	1	0	2	0	0	0	0	0	0	1	0	1	0	0	0	0	0
GUI/LOI	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2	2	2	1	1	0	1	0
OCL	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	3	1	0	1
Algebraic	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	1	0	2	0	0	0	1	0
Documentation	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1
Logs, Statecharts and State Machines	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	2	0
Object Z	0	0	0	0	0	0	0	0	0	1	2	0	0	0	0	1	0	0	0	0	0	0	0
Temporal Logic	0	0	0	0	1	0	1	0	0	0	0	0	1	1	2	2	0	2	0	1	0	0	0
Others	0	0	0	1	2	0	1	0	1	2	0	1	0	4	0	0	0	3	2	3	2	4	6

Figure 3.6: Publications by Year (Specification-Based Oracles)

From the specifications with more than two publications, Z Notation and OCL have been the most referenced both in absolute numbers and in distribution over the years. Three articles about Z (37,50%) were written by the same authors.

OCL (Object Constraint Language) is an extension of UML proposed by 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.

The first two studies on the use of state machines as oracles belong to the same authors and represent 40% of the total, from 2002 to 2003.

All articles about Object Z belong to the same research group.

From the 11 articles that discuss specifications with support for temporal logic, we could list the following languages: **MITL** (*Metric Interval Temporal Logic*) (Wang et al., 2005), **TRIO** (Hakansson et al., 2003) (Lin e 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., 2009); one article (Lin, 2007) references **ESML** (*Embedded Systems Modeling Language*) and **SIML** (*System Integration Modeling Language*); Lin e Ho (2000) use **time Petri nets**.

Table 3.6 represents the percentage of articles published in the last five and two years respectively. There was no published articles about Z notation in the last five years. There was only one article on algebraic specifications published in the last five years. And there were no published articles on Object Z in the last seven years.

Table 3.6: Last Years Publications

	Total	Five Years	Two Years
Temporal Logic	11	09,10%	00,00%
GUI/LOI	10	30,00%	10,00%
OCL	8	62,50%	12,50%
Z Notation	8	0,00%	0,00%
Algebraic	7	14,29%	14,29%
Documentation	5	60,00%	40,00%
State Machines	5	60,00%	40,00%
Object Z	4	00,00%	00,00%
Others	32	53,13%	31,25%

Z Notation

Richardson et al. (1992) present a work in progress to derive oracles based on paradigms of multiple specifications (RTIL and Z Notation) for the verification of test results in reactive systems. The authors consider the mapping from the specification name space

to the oracle name space and from the oracle name space to the implementation name space. The approach consists of three phases: deriving the oracle from the specifications, monitoring the test execution and applying the oracle procedure to the execution profile. The approach is meant to be mostly automated, besides there is no mention of tools to support it.

Real-Time Interval Logic (RTIL) supports the specification of orders of events with time constraints. Z Notation is used for scheduling tasks and events. It is given as example an elevator system. A temporal property supported by RTIL is "the elevator should not move with the doors open, so the door should not be opened in the interval between start and stop the elevator". Z notation, in the same example, is used in the scheduling of the elevators and call assignments, such as the presented scheme, *Best Elevator*, which defines what elevator should attend a call, if more than one is available.

Jia (1993) proposes the use of Z Notation as specification language for use as an model-based oracle. Should be given the following elements: the specification, the source code and the retrieve functions. These functions map the implementation in relation to the specification. A compiler generates a test driver from the specification and the retrieve functions. The source and driver are compiled to generate a executable tester, which reads a sequence of test cases and generates reports on the results of the execution, as such as the coverage.

Stocks e Carrington (1993) and Stocks e Carrington (1996) uses Z Notation to compose test templates and oracles, which are part of a framework. This framework is a formalism to define and organize specification-based testing. Template is a concept that represents the input and output spaces. An input space of an operation is the space from which the input can be represented, and it is defined as the restriction of the signatures of operations for the input components. A valid input space is the subset that meets the operation pre-conditions. The same concept applies to output space and valid output space but upon the signatures of the operation outputs. In this framework, the mapping between specification and implementation is done manually with the aid of tables that list the procedures, functions and calls to Z Notation schemes. A scheme is a block that describe static and dynamic aspects of the system, as states and operations.

Luqi et al. (1994) idealize the creation of an oracle based on Z Notation expressed in Latex and present a brief example of an operation of transferring values from one account to another.

Wang et al. (2003) discuss the creation of test oracles for multi-agents, with the use of Z Notation. The concepts of soft gene, role and agent are defined. The soft gene is an entity that has a set of behaviors and attributes. Roles are social behaviors expected from an individual and agents are entities that have some soft genes with some bound roles.

Agents can change their roles dynamically. Thus, a finite automaton can represent the role transitions of an agent. Two types of events may occur: linking or unlinking an agent to/from a role. Using an adaptation of the Z Notation (semi Z Notation), it is possible to represent the automaton itself and the states of the agent.

Miller e Strooper (2003) exemplify the use of oracle with a defined set of integers called IntSet. The specification of this set is presented in Z Notation and consists of eight schema (*state*, *init*, *add*, *remove*, *size*, *hasMore*, *isMember*, *next*), and the corresponding interface in Java. The *init* scheme represents the constructor method.

An *Animator* class serves as a communicator between the oracle and Possum (Hazel et al., 1997), an specification interpreter. This class has a method, *schemaCheck*. The oracle class inherits from the class under test and has a method called *abs* (abstraction) that receives the actual state of the class under test and returns the representation of the state in an abstract level. Because the implementation does not always support the same abstraction data types of the specification the author created a Java toolkit to support the abstract types of Z Notation on Java. Violations of pre-conditions must generate exceptions in the executions.

Coppit e Haddox-Schatz (2005) investigate the feasibility of revealing failures, adding assertions into the code, based on specification. It is given an example of an specification in Z Notation and Object Z. In one of the test cases, the assertions were expressed in Jass, a pre-processor that translates them into Java. The generated code corresponds to approximately 25% of total lines in the implementation.

The Z Notation can be applied to specify a broad spectrum of systems. When it is required the representation of the time concept, as the temporal properties in Richardson et al. (1992), the use of another language or adjustments are necessary. The states or concrete data from the implementation are mapped to a higher level of abstraction and compared by an oracle procedure that must first interpret the specification.

LOI (Level of Information)

LOI means *level of information* and it is a classification of oracle information and oracle procedure in relation to levels of detail. It is applied on the construction of oracles for testing GUI applications (Graphical User Interface).

According to Shahamiri et al. (2009), the internal behavior of the GUI is modeled using a representation of GUI elements and their actions. A formal model consists of GUI objects. Their specifications has its designs based on GUI attributes and they are used as oracles. Actions are defined by pre-conditions and effects.

Memon oriented a thesis (Xie, 2006a) and, together with collaborators, published eight articles about oracle for GUI. In Memon et al. (2000), it is presented an oracle expressed as a set of objects, object properties and actions. Given the formal model and a test case, such an oracle automatically derives the expected state for every action in the test case.

$s_j = [s_i, a_l; a_2; ...; a_n]$ denotes that the execution of a sequence of legal action $a_1...a_n$ started at a state s_i leads to a state s_j of an object. The modeling of the actions of the GUI are performed with the use of operators like *(*<Name, Pre-conditions, Effects>). As an example, it is given the operator for the action *set-background-color*:

Name: set-background-color(wX: window, Col: Color)
Pre-conditions: is-current (wX), background-color(wX, $oldCol$), $oldCol \neq Col$
Effects: background-color(wX, Col)

In this example, the operation of changing the background color must comply with the following pre-condition: *the new color must be different from the old color and the effect is that the window acquires the new color.*

The automated oracle uses the operators defined by a designer to derive the expected state corresponding to a given test case. An expected state S_1 is obtained from S_0 ($S_1 = [s_0, a_1]$). A state S_2 is obtained from S_1 ($s_2 = [s_1, a_2]$), and so on until all the expected sequence of states is derived. Once the expected result is derived, they are compared manually with the implementation execution or compared by means of an execution monitor.

Memon et al. (2003b) propose 11 types of oracles based on the combination between different levels of oracle information with different levels of oracle procedure. The authors compare the application of these types through four systems. As a result, it is shown (i) the time and space required by the oracles; (ii) that failures are detected early in the testing process when used detailed oracle information and complex oracle procedures, although at a high cost per test case; (iii) and the conclusion that the use of more expensive oracles results in detecting a large number of failures with relatively lower number of test cases.

Examples of oracle information are given in Memon et al. (2003a):

- LOI1 (complete): all properties of all objects of all windows in the GUI;
- LOI2 (complete visible): all properties of all the objects of all the visible windows in the GUI;
- LOI3 (active window): all properties of all the objects of the active window;
- LOI4 (widget): all properties of the object in question of the active window.

The same article presents the DART framework, addressed to systems that require frequent and automated regression testing. The article focuses specifically on smoke tests.

In Memon e Xie (2004b), the focus of the article is the fact that errors may appear and disappear at various points during execution of test cases. The authors raised two types of errors: transient, those that disappear until the end of the test case execution, and persistent, those that persist until the end of the test case execution. They observed that many errors are transient.

Memon e Xie (2004a) presented an empirical evaluation performed on oracles. The authors cite that short smoke tests in GUI oracles are effective at detecting large number of failures; there are classes of failures that the applied tests can not detect; short smoke tests execute a large amount of code; the whole process of smoke testing is feasible in terms of execution time and storage space.

Memon e Xie (2005) executed tests in an office open-source suite, the TerpOffice.

In Xie e Memon (2007), definitions of different levels of information (LOI) differ slightly from Memon et al. (2003a). The LOI is divided into three. The oracle procedure can act after each test case event or just after the last event. The authors split the three levels of information into six, depending on how the oracle procedure will work.

OCL

OCL (Object Constraint Language) is an extension of UML proposed by OMG, to allow the definition of constraints. OCL can be used, for example, to set limits of values to variables, and creating pre and post-conditions of methods.

Briand e Labiche (2001) and Briand e Labiche (2002) show a testing system called Totem composed by eight activities. The last article is a more detailed version of the first, containing 15 pages of appendix with UML. The activities are: checking completeness, correctness and consistency of the analysis model (A1); deriving dependence sequences from the use cases (A2); deriving requirements from the sequence diagrams of the system (A3); deriving test requirements from the class diagrams of the system (A4); deriving sequences of variants (A5); deriving requirements for system testing (A6); deriving test cases for system testing (A7); and deriving test oracles (A8).

Based on the logic of business process, some use cases should be performed before others. Sequences of use cases should be described in an activity diagram and, after that, in a graph so that regular expressions are produced. And sequences of use cases are generated and become part of the test plan.

For each use case, there is a sequence diagram. These diagrams are also represented as regular expressions. Sequences of scenarios of use cases are derived for testing. Terms

are obtained. Each term can contain a number of conditions (path realization condition) associated with their performance which should be expressed in a non-ambiguous way, in OCL. Defined the sequences of operations on each term, oracles are derived for each tested sequence. The main source used to derive test oracles are the post-conditions of operations in a sequence, defined in OCL.

Briand et al. (2003) analyze two measures of quality in contracts written in OCL used as oracles. The measures are *observability*, the probability that an error is detected by any of the components, and *diagnosability*, the ease in isolating an fault. The authors classify the level of detail of contracts (specifically in the post-conditions) into three parts: high precision, intermediate precision and low accuracy.

Pilskalns (2004) and Pilskalns et al. (2007) describe an approach to systematic evaluation of UML design models, incorporating the class and sequence diagrams in a combined model represented by a directed acyclic graph. It is built a CCT (Class and Constraint Tuple) from the class diagram and OCL expressions. The CCT and the directed graph are combined so that the nodes of the graph may contain OCL expressions, used as partial oracle. The OCL can be used on four levels: invariant for the primitive attributes of primitive types; invariant classes for classes and attributes that are classes; pre-conditions and post-conditions for methods.

Packevičius et al. (2007) advocate the approach of verifying the obtained output in relation to restrictions of imprecise OCL (imprecise OCL constraints), which can be viewed as expressions which define expected results within limits of possible values.

Skroch (2007) proposes a method to support validation activities for components. Sources for creating test oracles provide the requirements of the business domain. Lin (2007) uses OCL for development of automated models through model transformation approach.

Cheon e Avila (2010) propose an automated testing approach for Java programs by combining random testing and assertions in OCL. The OCL constraints are translated to runtime checks in AspectJ. The resulting aspect is called constraint checking aspect and it exists separated from the implementation code. The OCL constraints are translated to pointcuts and advices. Pointcuts define execution points and advices perform constraint checks. The authors cite the Dresden Toolkit (Demuth e Wilke, 2009), that can interpret OCL constraints on a UML model and generates runtime constraints checking code in AspectJ. They suggest adapting this tool for their proposed approach.

Algebraic Specification

Antoy e Hamlet (1992) address the problem of verifying the concordance between a formal specification of abstract data type (which can be used as an oracle) and its imple-

mentation. Specifically, the mapping between the states of the concrete implementation and the objects of the abstract specification.

The presented specification has a similar notation to several other algebraic specifications, such as *Act One* or *Larch*. Two codes are generated in C++: one generated by manual implementation (representing the *concrete world*) and one generated automatically by the specification (representing the *abstract world*). The implementation of self-checking is the union of both, with the addition of extra code to verify the agreement between the concrete and abstract worlds.

Yan (1999) focuses on object-oriented software testing. The authors argues that sometimes, comparing object internal representations may not be accurate even if they are observationally equivalent. First, he presents a scenario where two objects have the same internal representation and they executions are equivalent, as following:

$new\_i.push\_i(l).push\_i(2).push\_i(3).pop\_i().push\_i(4)$

Has as result an object with an internal representation $[1, 2, 4]$. And executing:

$new\_i.push\_i(l).push\_i(2).push\_i(4)$

Also leads to an object with the same internal representation. Another example shows a scenario where two object internal representations may differ. Considering the execution of the following method sequences:

$new\_i.push\_i(l).push\_i(2).push\_i(3).pop\_i()$

$new.push\_i(l).push\_i(2)$

Each execution may result in objects and their respective internal representations as shown below:

$o1 = ([1, 2, 3, nil, ..., nil], 2)$

$o2 = ([1, 2, nil, ..., nil], 2)$

The first element, an array, represents an stack content and the second element represents the stack size (pointer). Both objects have different internal representation, but they are observationally equivalent. Thus, if an oracle compares the internal representation of the two objects it can report that both results are not equivalent when they already are. The author uses Breu's algebraic design methodology (Breu, 1991) to resolve this issue. The steps are: after choosing two equivalent ground terms (as sequences derived from the class specification) as test case: (1) mapping the ground terms to its method sequences, which are implementation sequences; (2) executing the method sequences; (3) mapping the concrete results (the object internal representations) to abstract results (new ground terms); (4) using the axiom previously created to verify if the resulted ground terms can be written into a same normal form. If not, it means that an error has found.

Antoy e Hamlet (2000) propose a mapping from implementation states to abstract values, but inside the source code.

Zhu (2003) addresses the limitation of algebraic testing techniques in which the software is tested as a whole, without any application of integration techniques. The proposal is to organize the algebraic specifications to match the structure of object-oriented systems. The algebraic specification equations are divided into groups that represent the classes of the system. Each module in the specification should represent the values of objects of a class and other modules, securely imported. This security is achieved by ensuring that the axioms of a module does not modify the semantics of the imported modules.

Bagge e Haveraaen (2009) outline how axioms can be interpreted as rewrite rules and test oracles, imbuing the program code via concepts. The idea of concepts is allowing programmers to place restrictions on parameters of template. It is created a concept (in the given example, in SDF2 notation) with axioms. If an operation of a class in C++ satisfies this concept then the concept's name is attached to the signature of the operation.

Module Documentation

Peters and Parnas (Peters e Parnas, 1994) (Peters e Parnas, 1998) proposed the automatic oracle generation from relational program specification, called *Limited Domain Relations* (LD-Relations). An LD-Relation is a pair $< R_L, C_L >$, where R_L is an ordinary relation (may be represented as a set of initial and final acceptable states, or a function in a deterministic program) and C_L is a subset of R_L domain, known as a competence set (may be the set of initial states for which the program must terminate). A program P satisfies the specification if and only if:

- When started in any state x , if P ends, it does at the y state such that $< x, y >$ is an element of R_L, and;
- For all initial states x in C_L, P always ends.

The authors use tabular expressions in conjunction with auxiliary predicates as a representation of functions, user definitions, relations and auxiliary predicates. A user definition is a sequence of texts in a syntax in the programming language that is used to declare data structures, functions or symbols that are used in the specification and are not primitive to the programming language. Auxiliary predicates are expressions of predicates with names.

By providing an input and output data, the oracle returns true if the pair meets the specification or false, otherwise. The implementation of the oracle has four parts. One

for initializing internal data structures. The others return boolean values that validate the characteristic predicate of the competence set, domain and relation components.

Because it was a work in progress, the challenge was to convert the documentation into a form that could be executed. The authors suggest transforming the primitive relations, predicates, quantifiers and tabular expressions in C.

Peters e Parnas (2002) use the concept of monitor, a system that observes the behavior of a target system and reports whether the behavior is consistent with the requirements. This display, based on documented requirements, is the oracle. This article is intended for real-time systems.

Baharom e Shukur (2008) argue that gray-box testing approaches are usually based on knowledge obtained from the specification and source code, and rarely the design specification. The approach proposed by the authors uses the knowledge of the design specification in place of the source code. The specification was documented with Parnas's Module Documentation (MD) and it was used to generate the test oracle. The authors applied the concept of Limited Domain Relation to allow generation of oracles for non-deterministic programs, where the competence set contains the states in which the termination of the program is guaranteed.

Baharom e Shukur (2009) continue the research, focusing on the investigation of the use of Parnas's Module Documentation to automate the process of generating test oracles, specifically in the use of abstraction relation document as part of the approach. Abstraction relation is a part of the design document that maps from the concrete data structure to any possible sequence of events. The definition of abstraction relation is:

Let T be a set of traces and S, a set of data states. An abstraction relation (AR) is a subset of the cartesian product of T and S such that for each state of internal data in S there is at least one trace at T.

Alawneh e Peters (2010) present a tool that enhances an integrated development environment to give the user the ability to write formal specifications in a readable way and to generate test oracles automatically. They suggest that the specification may be expressed in mathematical notation as tabular expression and functions, derived from module internal design document. The specification is divided in the following components: constants, variables, auxiliar functions, predicate expressions and quantified expressions. As an example, they show a specification about a program that returns the greatest common divisor of two integers (Ggcd), presented in Table 3.7.

The program compares an integer i with an integer j. If i or j is lesser or equal to 0, the result is false. Otherwise, it is calculated the greatest common divisor of them by calling the auxiliary function *gcd* (defined in table 3.8). This function is recursive and will be called until $b = 0$.

Table 3.7: Ggcd Program Specification

Program Specification		
Boolean ggcd(**Integer** i, **Integer** j, **Integer** gcdvalue)		
	$i > 0 \wedge j > 0$	$i \leq 0 \vee j \leq 0$
gcdvalue =	gcd(i,j)	0
result =	TRUE	FALSE

Table 3.8: Auxiliary Function Definitions

Integer gcd(**Integer** a,**Integer** b) $\underline{\underline{df}}$	
$b \neq 0$	gcd(b, a%b)
$b = 0$	a

The oracle generation is similar to Peters e Parnas (1998), but the specification is written in OMDoc (Open Mathematical Documents), a content markup scheme for mathematical documents that can be used as a content language for the communication of mathematical software. The oracle is generated by a tool called TOG.

State Machines, Statecharts *and* Logs

In the Andrews e Zhang (2003) and Tu et al. (2009) approaches, a state machine description file (SMD) represents the specification of software under test. A parser uses SMD to generate the analyzer. The software creates a log in run-time of the events and sends it to the parser. The analyzer identifies whether there is a failure in the execution according to what would be expected by the state machine.

Andrews et al. (2002) propose the use of logs and oracles to identify the thoroughness of test cases by measuring how many transitions have been exercised in the analyzer.

Seifert (2008) uses UML state machines to automatically generate test cases and oracles, without the use of logs. A state machine specification is used to compute the correct observation sequences for certain entries. Then a acceptance graph is generated as test oracle, containing two possible nodes of acceptance, *pass* or *inconclusive*. If a sequence is not accepted by an automaton, called *acceptor*, the verdict will be a failure. To evaluate the approach, the authors implemented a set of tools called Teager, which automatically generates and executes test cases and executes the specifications of state machines.

Kanstren (2009) proposes the relation between test oracle and program comprehension (PC). PC is a field that deals with human understanding of software systems and its theoretical foundations are based on fields of study of human learning and understanding. Specifically, PC can make use of static information sources (artifacts of the program) and dynamic information sources (program execution). In an top-down approach, based

on the specification, a hypothesis is built on what is the program purpose. And, in a bottom-up approach, the program is examined to build a hypothesis on how it is operated. Finally, there is an attempt to combine the two hypotheses to determine whether the understanding is correct.

The propose of relating test oracle with program comprehension is given as follows: in the first step, the creation of the oracle information uses the program documentation and execution profiles as inputs to create the specification of what is expected from the software under test. Likewise, a functional hypothesis is constructed for the program, based on the specification. In the second step, the test monitor captures the information from the system execution (the Execution Profile). This information describes the behavior of the system. Similarly, the operational hypothesis is built to describe how the program behaves. Finally, the oracle procedure compares the model of the oracle information against the model of the Execution Profile to check the results. And the functional hypothesis is compared with the operational hypothesis. The authors proposed the following framework:

A model based on the software Execution Profile is built (as an EFSM). Automation is supported providing up ways to turn these models in test oracles. It is used a hybrid approach (top-down and bottom-up). First, models are built with bottom-up approach, i.e., from the EP. Then the user examines these models by a top-down approach analyzing the specification to determine if the model is correct and transforms it into oracle.

Object Z

Object Z is an object-oriented extension of Z Notation.

McDonald et al. (1997), McDonald e Strooper (1998) and MacColl et al. (1998) describe the derivation of test oracles from specifications in Object Z. There are three stages: optimization of the specification, translation of the optimized specification in a skeleton code in C++ and translation of the schema predicates in an implementation of C++. The steps of the test are: development of the testgraph, development of the test oracle, and development of the driver class. The testgraph is a subgraph of state/transition of the class under test and contains all states that must be reached by the test set.

McDonald et al. (2003) propose a passive oracle based on Object Z for testing classes in C++ and an implementation of the oracle called Warlock. The specification written in Object Z is used to create an abstract syntactic tree. Various constraints defined for the attributes of each node are checked to determine if the tree is properly formed. The tree is properly formed if the specification is correct with respect to the types and with respect to static constraints defined by the language. Then, the tree is optimized to simplify the translation into the implementation language. The code generator traverses the tree

recursively, generating constructors from the startup class schemes and member functions from operation schemes. The oracle is inserted into a wrapper class that inherits from the class under test (CUT) with the code used to check its behavior. This class is then tested in place of the CUT.

In the wrapper class the verification is done in abstract state space, not the concrete one. The passive oracle, then, consists of an invariant checker, a function to check the initial state, and verification functions for each member function defined in the CUT.

Temporal Logic

Temporal logic includes the design and study of specific systems for representation and comprehension of time (Venema et al., 1998).

Wang et al. (2005) present an approach for automatic generation of oracles for real-time systems based on MITL specification (Metric Interval Temporal Logic). From this specification, one can create a model in timed automata with accepting states (TAAS), witch is an automata that has clocks with two attributes: new and old. Their values do not change until some time designation exists in the current state. A sequence of timed states satisfies the specification if it can reach a final state of the automaton built from the specification. There are also other concepts of time like those presented in the following requirement of a Mars probe landing system: in the event of an error condition, the system must switch to emergency mode. The requirement description uses three variables, *Status*, *TimerInt* and *Done*, as shown below:

"When the *TimerInt* reaches the Control System and the reading of acceleration is not completed, the *status* should change to *Emergency* within $60ms$.

Their representation in temporal logic is:

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

The square-shaped symbol represents a "always in the interval" and the diamond means "sometime in the interval".

An automaton, in the example, was generated automatically based on temporal logic, with 9 states and 20 transitions. Given a timed state sequence, the oracle identifies whether it is according or not to the specification.

Lin e Ho (2000) correlate temporal logic formulas with Petri nets. Temporal logic describes the properties of the system due to its precise mathematical notations and Petri nets (operational language) are used to support the description of a system in terms of an abstract model that simulates its behavior.

Goldberg et al. (2005) discuss regression test for autonomous systems that operate without human interference for long periods, as the case of special probes and satellites.

In many cases, the oracle should not compare the exact outputs when performing an regression test. It is the case of planning the route between two points, as the given example in the soil of Mars. The oracle should identify whether the plan is acceptable and if it is generated within an acceptable time. That's because such problems are NP-complete and the search algorithm not necessarily gives the best way, but an acceptable way. With the plan (the output generated by the planner), the oracle performs its function on the basis of properties previously established. These properties are described by temporal logic in a framework proposed by the authors, the EAGLE.

Bouchet et al. (2008) present a method based on Lutess environment for testing interactive multimodal systems, in which various modes of the system can be used sequentially or concurrently, independently or combined. An example is given with a communication system that supports different modalities such as voice and gestures.

Lutess is a test environment that handles specifications in the Lustre language, which can be used as a programming language or specification language. A program is structured in nodes and each node consists of a set of equations that define outputs as functions of inputs. An expression is made of constants, variables and logical operators, arithmetic and language-specific notation. An example of the characteristic of temporal logic of Lustre is given by *OnceFromTo(A, B, C)*. The property A shall be maintained at least once between the instants when the events B and C occur.

Durrieu et al. (2009) present an Lustre specification-bases oracle approach, for the development of automated oracle applied to aeronautics (Airbus). They also presents the LETO tool.

Model Transformation

According to Mottu et al. (2008), model transformations are used on model-driven development with the objective of automating critical operations like refinement, code generation and refactoring. The paper gives a transformation framework in which there is a meta-model source, a transformation language and a meta-model target. A model is applied as input and must comply with the restrictions of the meta-model source. Additional pre-conditions can be applied before the processing. The result should be another model that must be in accordance with the post-conditions and constraints of the meta-model target. As an example, it is used the transformation of a class model into RDBMS model.

The authors state that three techniques can be used to implement oracles: model comparison, contracts and pattern matching. In the first one, a reference model (already

available or obtained) is compared with the resulting model of the transformation. For example, the tester provides a reference version of the transformation of the model. The reference transformation can produce the reference model from the test model. This reference model is compared with the output model. Contracts consist of pre and post-conditions. In pattern matching, a pattern is defined as a piece of model or a set of model elements. This technique consists in checking the presence of a pattern in a template.

Model Checking

According to Kuhn e Okum (2006), model checking is a formal technique based on state exploration. In their work, the input to a model checker has two parts: a state machine and temporal logical expressions over the states and execution paths. A model checker, conceptually, exercizes all reachable paths and verifies whether the temporal logic expressions are satisfied on all paths. If an expression is not met, the model checker generates a counter-example in the form of a sequence of states which points the non-satisfied expression.

In the context of software testing, a counter-example can be used as a test case with the input data and expected output that points an error in the implementation. Model Checking can be used with input data selection criteria, such as mutant analysis, or combinatorial testing.

The author cites Ammann et al. (1998) using mutant analysis. In this article, the authors propose the use of mutation operators to be applied in state machine and mutation operators to be used in the constraints. The model checker runs the mutants, one at a time, and counter-examples are generated when inconsistencies are found. When using a state machine mutant, a good implementation should diverge from the test performed with the same counter-example. These tests are referred to as failing tests. When using a temporal logic constraint mutant, the implementation should not correspond to the numbered sequence of states and outcomes. These are referred to as passing tests.

VDM

Aichernig (1999) explores possibilities for automation of black-box testing using the VDM formal method. This method is used as an oracle. A test framework is presented and it is based on the formal definition of abstraction as a homomorphism, i.e., a retrieve function that maps the concrete level (implementation) to the abstract level (VDM). If the retrieve function is implemented and the post-condition is executable, then the model can serve as a test oracle.

An approach to test automation is given. An implementation is executed with concrete input data. A retrieve function maps the concrete input data into abstract input data

and concrete output data into abstract output data. A pre-condition checker validates the input and feeds the oracle that checks the list to the produced output. If the post-condition evaluates to true, the test passed.

JML

Cheon e Leavens (2002) propose an oracle based on JML (Java Modeling Language), a specification language for Java interface behavior that has a checker assertions. The objective is to make writing test coding easier and maintainable.

With JML it is possible to represent pre-conditions, post-conditions, invariants, interfaces and intra-conditions. Post-conditions are divided into normal and exception. The former describes the behavior of a method when there is a return without an exception being thrown. The latter describes the behavior of a method when an exception is thrown.

The pre-conditions are treated in two different ways depending on the nature of the violation. The violations of pre-requisites for input are those that occur when a method receives an out of specification parameter. In such cases, one cannot conclude that there is a flaw in this method because the error was made by the client that sent the out of specification parameters. In such cases, the assertion verifier should not point failure on the outcome, but that the result is meaningless, since the parameters are not valid.

The violations of internal pre-conditions occur during the execution of the tested method body. That is, a method M calls an method F and sends invalid parameters. The method F throws an exception to M. The violation should be treated as a failure in M, because M is a client of F, being M the responsible for breaking the specification of F.

The specifications in JML are written in the Java class itself, between /@ @/ or after //@. The runtime assertion checker executes code in a transparent way (except for the sake of performance) unless a violation is found.

The Java code with JML specification is instrumented as follows: the original method becomes private and its name is changed (e.g.: addKgs becomes internal$addKgs); the verifier generates a new method with the same name as previously modified (addKgs) to takes its place, and calls the original method from within itself. The generated method first checks the pre-condition of the method and the invariant. If it finds violations, an exception is thrown. Then, within a try-block, the post-condition is treated (the normal post-condition if no exception, or the exception, otherwise). Upon post-condition violation, it is treated within a catch-block. In the finally-block, the invariant is checked again. To create test cases, it is used the JUnit.

Engels et al. (2007) propose an approach called model-driven monitoring, where the behavior of an operation is specified by a pair of diagrams. A class diagram describes

the static aspects of the system. The behavior of an operation is represented by visual contracts. The class "skeletons" are automatically created from the class diagram. Methods and assertions in JML are generated from the operations specified by the contracts. The method implementations need to be created manually, based on visual contracts. Assertions allow checking the consistency of models with the manually code generated.

Rajan et al. (2010) presents the application of jml-based oracles on Home Automation System. Gladisch et al. (2010) suggest the union of verification tools with capture and replay tools. The result was KeYGenU, a "chain-tool". The KeY system is responsible for the verification and test generation for a subset of Java and superset of Java Card. It is an automated theorem prover for first-order logic based on JML.

Other Specification

Brown et al. (1992) discuss the use of oracles based on code generated automatically by compiling specification developed in IORL (Input/Output Requirements Language). Given the formal specification, the programmers develop the code manually. With the same specification, a compiler generates code automatically. A set of test cases is performed in both implementations and the outputs are compared. The authors justify the use of automated coding as an oracle, not the final implementation, because it cannot be expected that the automatically generated code be efficient. Still, one cannot guarantee that the code be error free. However, its use is justified as an oracle because, even with errors, these errors are unlikely to be the same as the manually generated code. Thus, they can be used as pseudo-oracles. The IORL is a formal specification language based on graph and it was chosen by the authors for its availability and the maturity of its compiler.

Bieman e Yin (1992) propose an oracle from an executable specification language in which the syntax is similar to Lisp, called Prosper. The language was used to identify whether the output conforms to the post-condition. As an example, an oracle is given to a program for sorting lists in Lisp. The function to be tested is called Sort(). The post-condition defines that the output is an ordered permutation of the input. The oracle is composed by the following functions: IsOrdered(), which returns *true* if a list is ordered, and Permutation(), which receives two lists and returns *true* if one is a permutation of the other. The function SortPost() identifies whether a list $L2$ is a permutation of another $L1$ (by calling the Permutation()) and if $L2$ is ordered (by calling IsOrdered()). Finally, the oracle function SortOracle() specifies that the input should be a list, and the output of Sort() must be in accordance with the characteristic function produced by SortPost().

Grieskamp et al. (2001) describe techniques for obtaining oracle in ASML (Abstract State Machine Language), from use cases.

Xing e Jiang (2009) use the specification language TTCN-3 to define test oracles for GUI. With this language, it is possible to define a GUI with its objects, properties of objects, actions, states, oracle and test cases in structures similar to C++ structs. The language provides support for timers with several operators as start time and timeout.

Dan e Aichernig (2005) present the RAISE method and a modular language specification, RSL (Raise Specification Language). *Schemes* are basic units for building modules by means of expressions. The module is a specification based on models that are developed based on algebraic specification.

Li et al. (1997) present a method, *And-State*, for creating test oracle applicable to non-deterministic systems. The specification used is the SDL (Specification and Description Language), which may include more than one EFSM (extended finite state machine). Once the system is specified, an algorithm is responsible for transforming it into a model of oracle.

Meyer et al. (2007) present a unit testing framework that automates oracles using contracts in Eiffel. Contracts state what conditions the software must achieve and can be evaluated at runtime. They consist of pre-conditions, post-conditions and invariants.

D'Souza e Gopinathan (2006) consider the use of CTG (Complete Test Graphs) for hierarchical specifications. These specifications are particularly useful in describing large systems by allowing different levels of detail. The node of a hierarchical finite state machine represents a state or another state machine. A complete test graph is a structure that represents all possible test cases, given a purpose and a specification test. CTGs can be generated automatically by tools such as TVG, given a specification from a finite state machine and a test purpose.

It is given as an example, a system of coffee and tea. Assuming that the purpose is to test the coffee dispensing, it is possible to compute a CTG from a formal deterministic specification as a state machine. If the transaction between states is the output of coffee, the test passed. If it returns tea, the output is inconsistent. If the response is not tea or coffee, the test failed. From this CTG, test cases are generated automatically. The authors propose an algorithm as a test oracle that avoids the space overhead associated with the CTG.

Lozano et al. (2010) present the application of Constraint Programming as oracles on financial market systems. They chose Gecode as environment to write the constraints on the case study. But the paper focuses on how to create constraints from a comercial auction system using mathematical notation instead of using the Gecode/C notation.

3.3.1.2 Metamorphic Relation Based Oracles

Murphy (2008) presents the concept of metamorphic testing: "although it may be impossible to know whether the output of an application is correct for a particular input, these applications often exhibit properties such as if an input or system state is modified on a certain way, it can be predicted the new output, given the original output". A metamorphic relation expresses these properties.

Chen et al. (2001) present a method that combines metamorphic testing and fault-based testing using real and symbolic inputs. The symbolic test key is to represent infinitely many alternatives by a single symbolic alternative (Morell, 1990).

As an example, it is given a line of a program p?

$x := x * y + 3$

The result will, later, be multiplied by 2. Thus, the program must calculate the function $f(x, y) = 2xy + 6$.

One can replace the number 3 by another constant F in a program p':

$x := x * y + F$

F denotes all possible alternatives for the constant 3. Therefore, p' represents infinitely many alternative programs for p.

Using $x = 5$ and $y = 6$, for example, the result should be 66 (according to the given function). Note that the result of the program will also be 66. Using the program p', the goal is to find all constants F for $(30 * F) * 2$ such that p' gets the same result for p, i.e., where $(30 * F) * 2 = 66$. What, in this case, implies $F = 3$.

This proves that the test case with entries 5 and 6 differs from the original program p for all mutants constructed by replacing the constant 3 by any other constants.

Regarding the real inputs, it is given the following example: for $f(x, y) = 2 * x * y + 6$, $f(x, y) + f(-x, y)$ is always equal to 12 (it is not given a method to find such a relationship in the article). Several pairs of test cases (x, y) and $(-x, y)$ can be generated automatically. If $f(x, y) + f(-xy)$ do not result in 12 for at least one test case, an error must exist.

In the case of symbolic inputs, the test case $(5, 6)$ must result in $(60 + 2F)$. The goal is to solve, for the value(s) of F, in which the program satisfies the expected relation $f(x, y) + f(-x, y) = 12$. Since $(60 + 2F) + (-60 + 2F) = 12$, One can obtain $F = 3$. In conclusion, all alternative programs built in replacing 3 by another constant were eliminated by using the metamorphic test cases (5,6) and (-5,6). Another example is given as a function of area calculation.

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. (2003a) give the same concept applied on Chen et al. (2001), but with an example of a function for calculating power.

Gotlieb e Bernard (2006) apply 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 e Guderlei (2006b) use seven metamorphic relations to test on image processing operation by way of Euclidean distance transformation.

Mayer e 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 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 article, 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. The number of metamorphic relations and assertions found varied significantly among students. 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.

3.3.1.3 Neural Network Based Oracles

Vanmali et al. (2002) use an algorithm of backpropagation into a set of test cases applied to the original version of a system. The authors state that the trained network can be used as an oracle to evaluate the correctness of the output produced by new versions of the software and can be used as a simulated model, even though that model cannot guarantee 100% correction over the original program.

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 and non-equilateral.

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 are, 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. It is used backpropagation.

Lu e Ye (2007) use RBF (Radial Basis Function) for construction of oracle similarly to Mao et al. (2006b).

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. Higher thresholds can make the oracles point a faulty output as correct. In this sense, as higher the threshold, more the chance of faulty outputs be classified as expected outputs, therefore the oracle accuracy may decrease. Lower thresholds can make the oracle point a correct output as a faulty one.

3.3.1.4 N-Version Based Oracles

According to Shahamiri et al. (2009), N-Version is based on several implementations of the program, developed independently, and with the same functionality of the software under test. These versions are used as oracles. If there is disagreement about the output in the versions, the decision is based on voting and the most common values are used as the expected output.

Shimeall e Leveson (1988) use the N-Version concept on programs written in Pascal from a specification for a problem of combat simulation. Manolache e Kourie (2001) claim that M-mp, a variation of N-Version, provides low cost based on the justification that the program model do not need to be equivalent to the main program, but just the functionality in which the cost of verifying the correction is high needs to be covered.

The idea of comparing results between two or more implementation can be extended to programs that already exists. A golden version of a program can be used as an oracle, for example in regression testing, component harvesting (Hummel e 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 e Atkinson (2005) 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 the built components.

3.3.2 What are the limitations observed in using these oracles?

This subsection discusses the limitations of the use of test oracles found in the selected articles.

3.3.2.1 Limitations of Formal Specification-Based Oracles

Nadeem e Jaffar-ur Rehman (2005) point that for all that kind of oracles, if the specification is incorrect, then demonstrate 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 e 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 e 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 is an implementation, it may also contain errors. Oracles can reduce performance when embedded in the code, but remove them after the test phase can cause unexpected problems such as changing the behavior over time, which may be critical for real-time systems.

According to Peters e 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.

Still, it is possible to write a specification to which the oracle does not finish or does not do it in reasonable time. Non-termination can occur through endless recursion in the predicate or auxiliary function.

Peters e Parnas (1998) elicited difficulties encountered on implementing their approach in an application that searches, adds and removes elements in a hash table. Among them: (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 the trade-off between the specification precision and the simplicity; (ii) A test harness is a non-trivial program, also needing to be checked carefully; (iii) Finally, not all program behavior 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 occur an explosion in the number of states, 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 corretitude to validate it as an oracle. The author cites the lack of empirical studies on the use of state machines.

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

According to Ammann et al. (1998), in the approach with the use of model checking, results must be deterministic to compare model and implementation.

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 e Carrington (1993) advocate the use of formal specification with Z Notation, but they do not discuss the problems of producing a oracle procedure. According to

Gotlieb e Bernard (2006) and Gargantini e Riccobene (2001), a limitation is the high cost relative to the time spent in developing a specification based on notations such as Z. Coppit e Haddox-Schatz (2005) show that the expressiveness of the assertion language can significantly affect the cost of implementing the assertions. 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 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.

Briand et al. (2003) indicate that, in their case study, there were found 20% less failures with the use of contracts as oracles than with the mutant test.

3.3.2.2 Metamorphic Relation Based Oracle Limitations

According to Chen et al. (2003a), metamorphic relations that cause higher "difference between executions" tend to be better. But this concept was not explicitly set, covering all aspects of the executions of the program. More research should be conducted to provide more explicit guidelines.

According to Chan et al. (2006), the choice of metamorphic relations was based on experience of the testers.

Murphy et al. (2009b) describe that metamorphic testing can be a manually intensive technique for more 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 $ShortestPath(A, C) = ShortestPath(A, B) + ShortestPath(B, C)$.

3.3.2.3 Neural Network Based Oracle Limitations

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 may not be easy, as the amount of layers and neurons. How to select the training set from the test cases is another key problem that must be considered carefully.

Lu e 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) observe about the relation between the chosen threshold value and the neural network accuracy. It seems that there is nofinal answer to one choose the network initial settings and it may vary between applications or application domains.

3.3.2.4 N-Version Based Oracle Limitations

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 e Leveson (1988) mention that N-Version is not a substitute for functional tests. In the experiment, N-Version did not tolerate many of the faults detected by other techniques to eliminate failure.

3.3.3 There are tools that support oracles for dynamical systems?

There were considered as "tools" any means of automation associated with oracles, including specification language translators, development environments that support oracles and frameworks. Table 3.9 presents the list of tools and their descriptions.

Table 3.9: Oracle Support

Name	References	Description
T-Vec	(Kuhn e Okum, 2006)	Development environment with associated specification and verification method for critical systems (temporal logic).
LETO/ Ocasime	(Durrieu et al., 2009)	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).
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).

Name	References	Description
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.
TAGS	(Brown et al., 1992)	Compiles IORL specification in Ada.
TOG	(Peters e Parnas, 1994) (Alawneh e Peters, 2010)	Test oracle generating tool, from a relational specification of the program and tabular expressions.
DSMDiff	(Lin, 2007)	Computes the difference between specific domain models, in model transformation.
TOM	(Silva et al., 2008)	Generates oracle specification based on state machines.
PLASMA	(Goldberg et al., 2005)	Route plan generation system, based on model. A real-time verification based oracle is proposed for this system.
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.
Extreme Harvesting	(Hummel e Atkinson, 2005)	Tool to find and collect pre-fabricated components for reuse from the Internet.
Protest	(Hoffman e Strooper, 1991)	Set of Prolog programs that support the development of test scripts and their applications to test modules implemented in C.
BZTT	(Miller e Strooper, 2003)	Tool that uses the specification to generate states for testing. It uses constraints solver to search for a sequence of operations that reach every state.
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).
PATHS	(Memon et al., 2000)	GUI testing tool with support to AI and formal model test oracles.

Name	References	Description
MD-TEST	(Baharom e 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.
TEAGER	(Seifert, 2008)	Test environment that allows execution of state machines specifications.
MaC	(Xie e Memon, 2007)	Runtime monitoring tool.
JPaX	(Xie e Memon, 2007)	Runtime monitoring tool.
TAOS	(Richardson, 1994)	Test tool with support to GIL specification based manual oracles.
CaslTest	(Machado et al., 2005)	Test tool with support to Casl specification based oracles.
USE	(Pilskalns et al., 2007)	Validation tool that checks the states of objects generated from class diagrams in relation to the OCL.
TROT	(Hagar e Bieman, 1996)	Testing tools that support the creation of test oracles. They check the corretitude of the equation implementation, based on Anna formal specification language.
TOTEM	(Briand e Labiche, 2001)	System test methodology based on UML in which the information is derived from OCL.
ORSTRA	(Xie, 2006b)	Support tool that checks results from regression test.
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.
Dresden OCL Toolkit	(Cheon e Avila, 2010)	Interprets OCL constraints from a UML model and generates AspectJ code.
NeuronDot-Net	(Shahamiri et al., 2010)	Engine which can be used to build different types of neural networks.

Name	References	Description
KeYGenU	(Gladisch et al., 2010)	Chain-tool. KeY is a static checker that can automatically prove properties. GenUTest is a capture and replay tool

3.4 Quality Criteria Aplication

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

Table 3.10: 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	Neural Networks	N-Version or similar	Limitations
Aggarwal et al. (2004)									√		√
Aichernig (1999)							√				√
Aichernig et al. (2009)							√				√
Alawneh e Peters (2010)							√				
Almog e Heart (2010)						√					
Andrews et al. (2002)							√				
Andrews e Zhang (2003)							√				√
Antoy e Hamlet (1992)							√				√
Antoy e Hamlet (2000)							√				√
Bagge e Haveraaen (2009)							√				√
Baharom e Shukur (2008)							√				
Baharom e Shukur (2009)							√				√
Bieman e Yin (1992)							√				
Bouchet et al. (2008)				√			√				
Briand e Labiche (2001)							√				
Briand e 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. (2001)								√			√
Chen et al. (2002)								√			√
Chen e Subramaniam (2002)							√				√
Chen et al. (2003a)								√			√
Chen et al. (2003b)								√			√
Cheon e Leavens (2002)							√				
Cheon (2007)							√				
Cheon e Avila (2010)							√				
Cho e Lee (2005)		√					√				√
Coppit e Haddox-Schatz (2005)							√				√
Dan e Aichernig (2005)							√				√
Ding et al. (2010)								√			√
Durrieu et al. (2009)		√	√	√			√				
D'Souza e Gopinathan (2006)							√				
Edwards (2001)							√				
El Ariss et al. (2010)										√	
Engels et al. (2007)							√				

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Neural Networks	N-Version or similar	Limitations
Gargantini e Riccobene (2001)							√				
Gladisch et al. (2010)							√			√	√
Goldberg et al. (2005)		√	√	√			√				
Gotlieb e Bernard (2006)								√			
Grieskamp et al. (2001)							√				
Hagar e Bieman (1996)							√				√
Hakansson et al. (2003)				√			√				
Hoffman e Strooper (1991)							√				
Hu et al. (2006)								√			√
Hummel e Atkinson (2005)										√	√
Jia (1993)							√				√
Jin et al. (2008)									√		√
Jin et al. (2009)									√		√
Jonsson e Padilla (2001)							√				
Kanstren (2009)							√				
Kim-Park et al. (2010)							√				√
Kuhn e Okum (2006)							√				√
Kuo et al. (2010)								√			√
Li et al. (1997)							√				√
Lin et al. (1997)							√				√
Lin e Ho (2000)				√			√				
Lin e Ho (2001)				√			√				√
Lin (2007)		√	√	√			√				
Lozano et al. (2010)							√				
Lu e Ye (2007)									√		
Luqi et al. (1994)							√				
Machado et al. (2005)							√				√
MacColl et al. (1998)							√				
Manolache e Kourie (2001)										√	
Mao et al. (2006b)									√		√
Mayer e Guderlei (2006b)								√			√
Mayer e Guderlei (2006a)								√			
McDonald et al. (1997)							√				√
McDonald e Strooper (1998)							√				√
McDonald et al. (2003)							√				√
Memon et al. (2000)							√				
Memon et al. (2003b)							√				√
Memon et al. (2003a)							√				
Memon e Xie (2004b)							√				
Memon e Xie (2004a)										√	√
Memon e Xie (2005)							√				√
Memon et al. (2005)							√				√
Meyer et al. (2007)							√				
Miller e Strooper (2003)							√				
Mottu et al. (2008)							√				√
Murphy (2008)								√			√
Murphy et al. (2009a)								√			√
Murphy et al. (2009b)								√			√
Nadeem e Jaffar-ur Rehman (2005)							√				
O'Malley (1996)							√				√
Packevičius et al. (2007)							√				
Peters e Parnas (1994)							√				√
Peters e Parnas (1998)							√				√
Peters e Parnas (2002)							√				
Pilskalns (2004)							√				
Pilskalns et al. (2007)							√				√
Rajan et al. (2010)							√				√
Richardson et al. (1992)				√			√				
Richardson (1994)				√			√				√
Seifert (2008)		√					√				
Shahamiri et al. (2009)						√			√	√	√
Shahamiri et al. (2010)									√		√
Shimeall e Leveson (1988)										√	√
Shukla et al. (2005)							√				√
Silva et al. (2008)							√				√

	Criterion 1				Criterion 2	Criterion 3	Criterion 4				Criterion 5
	Dynamical	Embedded	Critical	Real-Time	Simulink Scicos	Classification Comparison	Specification	Metamorphic Relations	Neural Networks	N-Version or similar	Limitations
Skroch (2007)							✓				
Stocks e Carrington (1993)							✓				✓
Stocks e Carrington (1996)							✓				
Taneja et al. (2010)										✓	
Tsai et al. (2005b)										✓	✓
Tsai et al. (2005a)										✓	✓
Tu et al. (2009)							✓				✓
Vanmali et al. (2002)									✓		
Wang et al. (2003)				✓			✓				
Wang et al. (2005)			✓	✓			✓				
Xie (2006a)							✓				✓
Xie (2006b)										✓	✓
Xie e Memon (2007)							✓				✓
Xie et al. (2009)								✓			
Xie et al. (2010)								✓			
Xing e Jiang (2009)							✓				
Yan (1999)							✓				
Ye et al. (2006)									✓		
Yoo (2010)								✓			✓
Zhang et al. (2009)								✓			✓
Zhou et al. (2010)							✓				
Zhu (2003)							✓				

3.5 Final Remarks

This chapter presented the systematic review on the topic "oracles associated with models for dynamical systems", in which the goal is the identification of articles based on three items of interest: oracles applicable to dynamical systems, limitations on the use of automated test oracles applicable to dynamical systems and tools to support test oracles for dynamical systems.

We did not find works with emphasis in dynamical systems. Therefore, we considered any oracle that could be adapted to be applied on dynamical systems. We selected 125 articles being 90 works addressing specification-based oracles, 19 are based on metamorphic relations, 11 are based on N-Version or a similar approach and 10 in neural networks. From the specification-based, 6.4% refer to the Z Notation, 8.8% to temporal logic specifications, 6.4% OCL, 5,6% refer to algebraic specifications, 4.94% and 20.00% to other specifications with less than three 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 involve 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 another. There is difficulty in expressing a detailed specification considering the time and cost for that. The higher the expression, closer to the implementation is the amount of resources spent to 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. The use can be laborious for large systems. Regarding the oracles based on neural networks, 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 areas. Oracles based on N-Version are expensive given the need to create several versions of the system.

There were listed tools that support test oracles, among which stand out those that apply to embedded and real-time systems: T-VEC, LETO and LUTESS.

Chapter

4

Final Remarks

The main contribution of this technical report is presenting the overview of test oracles applicable to dynamical systems, as well as its limitations and the identification of tools to support oracles. For reach these goals, we conducted a systematic review in which the theme is "oracles associated with models for dynamical systems". We obtained the following considerations: there was no work specifically focused on dynamical system oracles until the end of the selection phase (Chapter 3, Section 3.1.5). From the oracles that can be applied to models for dynamical systems, nearly three quarters use formal specification as oracle information. Z Notation and OCL were the most used, although the former one has not being used for the last five years while OCL has been used with more frequency lately.

About the identified limitation, there is the difficulty in representing the system at different levels of abstraction and in mapping between these levels. The presence of errors in the information of the oracle, either in the specification, metamorphic relation, or in samples of neural networks training, causes loss of confidence on the comparison between the expected and obtained results.

The tools that support test oracles, we highlighted those that apply to embedded systems and real-time: T-VEC, LETO and LUTESS.

References

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.

Aghion, P.; Bacchetta, P.; Banerjee, A. A corporate balance-sheet approach to currency crises. *Journal of Economic Theory*, v. 119, n. 1, p. 6 – 30, macroeconomics of Global Capital Market Imperfections, 2004.

Aichernig, B. Automated black-box testing with abstract vdm oracle. 1999. Disponível em `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, 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.

Ammann, P.; Offutt, J. *Introduction to software testing*. New York, NY, USA: Cambridge University Press, 2008.

Ammann, P. E.; Black, P. E.; Majurski, W. Using model checking to generate tests from specifications. In: *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98*, IEEE Computer Society, 1998, p. 46–54.

Andrews, J.; Fu, R.; Liu, V. Adding value to formal test oracles. In: *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, 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: *Fourth International Conference on Computing and Information, 1992. Proceedings. ICCI '92.*, 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. Estrutura para utilização de cbir em oráculos gráficos. 2008.

Araujo, R. F.; Delamaro, M. E. Tetoods - testing tool for dynamic systems. In: *Tools Session – Brazilian Software Engineering Symposium*, Brazil: SBC, 2008.

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: *International Conference on Electrical Engineering and Informatics, 2009. ICEEI '09.*, 2009, p. 304–308.

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.

Bieman, J.; Yin, H. Designing for software testability using automated oracles. In: *Test Conference, 1992. Proceedings., International*, 1992, p. 900–.

Biolchini, J. C. 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.

Bouchet, J.; Madani, L.; Nigay, L.; Oriat, C.; Parissis, I. Formal testing of multimodal interactive systems, p. 36–52. 2008.

Breu, R. *Algebraic specification techniques in object oriented programming environments*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1991.

Briand, L.; Labiche, Y. A uml-based approach to system testing. 2001. Disponível em `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. Disponível em `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. Disponível em `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. 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.

Chen, J.; Subramaniam, 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. Oxford, cited By (since 1996) 6; Conference of 26th Annual International Computer Software and Applications Conference; Conference Date: 26 August 2002 through 29 August 2002; Conference Code: 59851, 2002, p. 327–333.

Chen, T.; Kuo, F.-C.; Tse, T.; Zhou, Z. Q. Metamorphic testing and beyond. In: *Eleventh Annual International Workshop on Software Technology and Engineering Practice, 2003.*, 2003a, p. 94–100.

Chen, T.; Tse, T.; Zhou, Z. Fault-based testing in the absence of an oracle. Chicago, IL, cited By (since 1996) 4; Conference of 25th Annual International Computer Software and Applications Conference (COMPSAC)2001; Conference Date: 8 October 2001 through 12 October 2001; Conference Code: 58826, 2001, p. 172–178.

Chen, T.; Tse, T.; Zhou, 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, 2003b.

Cheon, Y. Abstraction in assertion-based test oracles. In: *Seventh International Conference on Quality Software, 2007. QSIC '07.*, 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.
Disponível em `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.

Demuth, B.; Wilke, C. Model and object verification by using dresden ocl. In: *Russian-German Workshop Ïnnovation Information Technologies: Theory & Practice: Ufa, Russia*, 2009, p. 1020 –1025.

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.

D'Souza, D.; Gopinathan, M. Computing complete test graphs for hierarchical systems. In: *Fourth IEEE International Conference on Software Engineering and Formal Methods, 2006. SEFM 2006.*, 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*, 2009.

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. Disponível em `http://dx.doi.org/10.1007/978-3-540-69489-2_23`

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.

claude Gaudel, M. Testing can be formal, too. Springer-Verlag, 1995, p. 82–96.

Geromel, J. C.; Palhares, A. G. B. *Análise linear de sistemas dinâmicos*. São Paulo: Edgar Blücher, 2004.

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.

Gotlieb, A.; Bernard, P. A semi-empirical model of test quality in symmetric testing: Application to testing java card apis. In: *Sixth International Conference on Quality Software, 2006. QSIC 2006.*, 2006, p. 329–336.

Grieskamp, W.; Lepper, M.; Schulte, W.; Tillmann, N. Testable use cases in the abstract state machine language. In: *Second Asia-Pacific Conference on Quality Software, 2001. Proceedings.*, 2001, p. 167–172.

Guttag, J. Abstract data types and the development of data structures. *Commun. ACM*, v. 20, n. 6, p. 396–404, 1977.

Hagar, J. B.; Bieman, J. M. Using formal specifications as oracles for system-critical software. *ACM Ada Letters*, v. 6, p. 55–72, 1996.

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.
Disponível em `http://dx.doi.org/10.1007/s10009-003-0107-8`

Hazel, D.; Strooper, P.; Traynor, O. Possum: An animator for the sum specification language. *Asia-Pacific Software Engineering Conference*, v. 0, p. 42, 1997.

Hoffman, D.; Strooper, P. Automated module testing in prolog. *IEEE Transactions on Software Engineering*, v. 17, n. 9, p. 934–943, 1991.

Hu, P.; Zhang, Z.; Chan, W.; Tse, T. An empirical comparison between direct and indirect test result checking approaches. Portland, OR, cited By (since 1996) 3; Conference of 3rd International Workshop on Software Quality Assurance, SOQUA 2006, co-located with the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM SIGSOFT 2006 / FSE 14; Conference Date: 6 November 2006 through 6 November 2006; Conference Code: 70040, 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.

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. Disponível em `http://dx.doi.org/10.1007/3-540-48213-X_23`

Jost, J. *Dynamical systems : examples of complex behaviour*. Springer, 2005.

Kaner, C.; Falk, J. L.; Nguyen, H. Q. *Testing computer software, second edition*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

Kanstren, T. Program comprehension for user-assisted test oracle generation. In: *Fourth International Conference on Software Engineering Advances. ICSEA '09.*, 2009, p. 118–127.

Kim-Park, D.; de la 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.; de la 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*,).

Kitchenham, B. *Procedures for performing systematic reviews technical report tr/se-0401*. Relatório Técnico, Keele University and NICTA, 2004.

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.

Li, J.; Liu, H.; Seviora, R. Constructing automated protocol testing oracles to accommodate specification nondeterminism. In: *Sixth International Conference on Computer Communications and Networks, 1997. Proceedings.*, 1997, p. 532–537.

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.* Tese de Doutoramento, 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.

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: *Second International Conference on Formal Engineering Methods, 1998. Proceedings.*, 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.

Maler, O. A unified approach for studying discrete and continuous dynamical systems. In: *Proceedings of the 37th IEEE Conference on Decision and Control, 1998.*, 1998, p. 2083 –2088 vol.2.

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: *5th IEEE International Conference on Cognitive Informatics, 2006. ICCI 2006.*, 2006a, p. 517–522.

Mao, Y.; Boqin, F.; Li, Z.; Yao, L. Neural networks based automated test oracle for software testing. 2006b.
Disponível em `http://dx.doi.org/10.1007/11893295_55`

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: *Sixth International Conference on Quality Software, 2006. QSIC 2006.*, 2006b, p. 85–92.

McDonald, J.; Murray, L.; Strooper, P. Translating object-z specifications to object-oriented test oracles. In: *Software Engineering Conference, 1997. Asia Pacific ... and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*, 1997, p. 414–423.

McDonald, J.; Strooper, P. Translating object-z specifications to passive test oracles. In: *Second International Conference on Formal Engineering Methods, 1998. Proceedings.*, 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.

Memon, A.; Banerjee, I.; Hashmi, N.; Nagarajan, A. Dart: a framework for regression testing "nightly/daily builds" of gui applications. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003a, p. 410–419.

Memon, A.; Banerjee, I.; Nagarajan, A. What test oracle should i use for effective gui testing? In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, 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: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004a, p. 8–17.

Memon, A.; Xie, Q. Using transient/persistent errors to develop automated test oracles for event-driven software. In: *19th International Conference on Automated Software Engineering, 2004. Proceedings.*, 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.
Disponível em `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: *First International Conference on Software Engineering and Formal Methods, 2003.Proceedings.*, 2003, p. 14–23.

Morell, L. J. A theory of fault-based testing. *IEEE Trans. Softw. Eng.*, v. 16, n. 8, p. 844–857, 1990.

Mottu, J.-M.; Baudry, B.; Traon, Y. Model transformation testing: oracle issue. In: *IEEE International Conference on Software Testing Verification and Validation Workshop, 2008. ICSTW '08.*, 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.; 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: *International Conference on Software Testing Verification and Validation, 2009. ICST '09.*, 2009b, p. 436–445.

Myers, G. *The art of software testing.* John Wiley and Sons, 2004.

Nadeem, A.; Jaffar-ur 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.

Neelin, J. D.; Latif, M.; Jin, F. Dynamics of coupled ocean-atmosphere models: The tropical problem. *Annual Review of Fluid Mechanics*, v. 26, n. 1, p. 617–659, 1994.

O'Malley, T. O. *A model of specification-based test oracles.* Tese de Doutoramento, 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.

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.* Tese de Doutoramento, Pullman, WA, USA, chair-Andrews, Anneliese, 2004.

Rajan, A.; du 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*,). Disponível em `http://doi.acm.org/10.1145/1865875.1865882`

Richardson, D.; Aha, S.; O'Malley, T. Specification-based test oracles for reactive systems. In: *International Conference on Software Engineering, 1992.*, 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.

Söderström, T. *Discrete-time stochastic systems.* London: Springer-Verlag, 2002.

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.; Mohd-Hashim, S. A comparative study on automated software test oracle methods. In: *Fourth International Conference on Software Engineering Advances, 2009. ICSEA '09.*, 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.

Shimeall, T.; Leveson, N. An empirical comparison of software fault tolerance and fault elimination. In: *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988.*, 1988, p. 180–187.

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: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2007. ECBS '07.*, 2007, p. 459–466.

Smeulders, A. W. M.; Worring, M.; Santini, S.; Gupta, A.; Jain, R. Content-based image retrieval at the end of the early years. *IEEE Transactions on Patterns Analysis and Machine Intelligence*, v. 22, 2000.

Stocks, P.; Carrington, D. Test templates: a specification-based testing framework. In: *15th International Conference on Software Engineering, 1993. Proceedings.*, 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.

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*,).

The Institute of Electrical and Eletronics Engineers Ieee standard glossary of software engineering terminology. IEEE Standard, 1990.

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, 2005.*, 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, 2009. 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.

Venema, Y.; (ed, L. G.; Guide, B.; Logic, P.; Publishers, B. Temporal logic. In: *The Blackwell Guide to Philosophical Logic. Blackwell Philosophy Guides (2001*, Basil Blackwell Publishers, 1998.

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: *10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings.*, 2005, p. 440–449.

Weyuker, E. J. On testing non-testable programs. *The Computer Journal*, v. 25, n. 4, p. 465–470, 1982.

Xie, Q. *Developing cost-effective model-based techniques for gui testing.* Tese de Doutoramento, 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.
Disponível em 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: *9th International Conference on Quality Software, 2009. QSIC '09.*, 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.

Xing, X.; Jiang, F. Gui test case definition with ttcn-3. In: *International Conference on Computational Intelligence and Software Engineering, 2009. CiSE 2009.*, 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 International Conference on Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings.*, 1999, p. 928–932 vol.1.

Ye, M.; Feng, B.; Zhu, L.; Lin, Y. Automated test oracle based on neural networks. In: *5th IEEE International Conference on Cognitive Informatics, 2006. ICCI 2006.*, 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.

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.

Zhao, X.-Q. *Dynamical systems in population biology.* Springer, 2003.

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: *Third International Conference on Quality Software, 2003. Proceedings.*, 2003, p. 91–98.